Red Hat JBoss Middleware for OpenShift 3
Red Hat JBoss Fuse Integration Services 2.0 for OpenShift

Installing and developing with Red Hat JBoss Fuse Integration Services 2.0 for OpenShift

Red Hat JBoss Middleware for OpenShift Documentation Team

Red Hat JBoss Middleware for OpenShift 3 Red Hat JBoss Fuse Integration Services 2.0 for OpenShift

---

Installing and developing with Red Hat JBoss Fuse Integration Services 2.0 for OpenShift

Legal Notice

Abstract

Guide to using Red Hat JBoss Fuse Integration Services 2.0 for OpenShift

Table of Contents

CHAPTER 1. BEFORE YOU BEGIN

1.1. RELEASE NOTES

See the Fuse Integration Services 2.0 for OpenShift Release Notes for important information about this release.

1.2. VERSION COMPATIBILITY AND SUPPORT

See the Red Hat JBoss Fuse Supported Configurations page for details of version compatibility and support.

1.3. SUPPORT FOR WINDOWS O/S

The developer tooling (**oc** client and Container Development Kit) for FIS is fully supported on the Windows O/S. The examples shown in Linux command-line syntax can also work on the Windows O/S, provided they are modified appropriately to obey Windows command-line syntax.

1.4. GETTING STARTED FOR ADMINISTRATORS

If you are an administrator, we recommend you read the instructions in the OpenShift Primer to get started with installing and administering a small OpenShift cluster.

CHAPTER 2. GET STARTED FOR ADMINISTRATORS

You can start using Fuse Integration Services by creating an application and deploying it to OpenShift using OpenShift Source-to-Image (S2I) application development workflows.

2.1. PREPARE THE VIRTUAL OPENSHIFT SERVER

1. Start the virtual OpenShift Server.
2. Log in to the virtual OpenShift Server as an administrator, as follows:

```
oc login -u admin -p admin
```

> **Note**
>
> The **`admin`** user is a standard account that is automatically created on the virtual OpenShift Server.

3. Install the FIS image streams. Enter the following commands at a command prompt:

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/GA
oc replace --force -n openshift -f ${BASEURL}/fis-image-streams.json
```

4. Install the FIS templates. Enter the following commands at a command prompt:

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/GA
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-camel-log-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-camel-rest-sql-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-cxf-rest-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-config-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-drools-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-infinispan-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-rest-sql-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-teiid-template.json
```

```
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-
boot-camel-xml-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-
boot-cxf-jaxrs-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-
boot-cxf-jaxws-template.json
```

CHAPTER 3. GET STARTED FOR DEVELOPERS

You can start using Fuse Integration Services by creating an application and deploying it to OpenShift using one of the following OpenShift Source-to-Image (S2I) application development workflows:

S2I binary workflow

S2I with build input from a *binary source*. This workflow is characterised by the fact that the build is partly executed on the developer's own machine. After building a binary package locally, this workflow hands off the binary package to OpenShift. For more details, see Binary Source from the *OpenShift 3.3 Developer Guide*.

S2I source workflow

S2I with build input from a *Git source*. This workflow is characterised by the fact that the build is executed entirely on the OpenShift server. For more details, see Git Source from the *OpenShift 3.3 Developer Guide*.

3.1. PREREQUISITES

3.1.1. Access to an OpenShift Server

The fundamental requirement for developing and testing FIS projects is having access to an OpenShift Server. You have the following basic alternatives:

- Section 3.1.1.1, "Install the Container Development Kit (CDK) on your local machine"
- Section 3.1.1.2, "Get remote access to an existing OpenShift Server"

3.1.1.1. Install the Container Development Kit (CDK) on your local machine

In most cases, the most practical alternative for a developer is to install the Red Hat CDK on their local machine. Using the CDK, you can boot a virtual machine (VM) instance that runs an image of OpenShift on Red Hat Enterprise Linux (RHEL) 7. An installation of the CDK consists of the following key components:

- A virtual machine (libvirt, VirtualBox, or Hyper-V)
- Vagrant (script for configuring and booting the VM image)
- Vagrant plug-ins (providing special features of the CDK)
- VM image of *OpenShift on RHEL 7*

Installing the CDK is a fairly lengthy and complex procedure. For FIS 2.0 you need to install version 2.3 of the CDK. Detailed instructions for installing and using the CDK 2.3 are provided in the following guides:

- Red Hat CDK 2.3 Installation Guide
- Red Hat CDK 2.3 Getting Started Guide

If you opt to use the CDK, it is recommended that you read and thoroughly understand the content of the preceding guides before proceeding with the examples in this chapter.

Important

OpenShift has fairly strict requirements for the versions of the CDK components you use. Make sure that you follow the guidance given in the *Installation Guide*. If necessary, you might have to uninstall an existing component and re-install the correct version.

3.1.1.2. Get remote access to an existing OpenShift Server

Your IT department might already have set up an OpenShift cluster on some server machines. In this case, the following requirements must be satisfied for getting started with FIS 2.0:

- The server machines must be running OpenShift Container Platform 3.3 (or later versions, as documented in the Release Notes).
- Ask the OpenShift administrator to install the latest FIS 2.0 container base images and the FIS templates on the OpenShift servers.
- Ask the OpenShift administrator to create a user account for you, having the usual developer permissions (enabling you to create, deploy, and run OpenShift projects).
- Ask the administrator for the URL of the OpenShift Server (which you can use either to browse to the OpenShift console or connect to OpenShift using the **oc** command-line client) and the login credentials for your account.

3.1.2. Java Version

On your developer machine, make sure you have installed a Java version that is supported by JBoss Fuse 6.3. For details of the supported Java versions, see Supported Configurations.

3.1.3. Install the Requisite Client-Side Tools

We recommend that you have the following tools installed on your developer machine:

Apache Maven 3.3.x

Required for local builds of OpenShift projects. Download the appropriate package from the Apache Maven download page. Make sure that you have at least version 3.3.x (or later) installed, otherwise Maven might have problems resolving dependencies when you build your project.

Git

Required for the OpenShift S2I source workflow and generally recommended for source control of your FIS projects. Download the appropriate package from the Git Downloads page.

OpenShift client

If you are using the CDK, the **oc** client tool can conveniently be installed as follows:

```
$ vagrant service-manager install-cli openshift
```

For more details, see Preparing Host System for Using OpenShift from the Command Line in the CDK 2.3 *Getting Started Guide*.

If you are not using the CDK, follow the instructions in the CLI Reference to install the **oc** client tool.

(Optional) **Docker client**

Advanced users might find it convenient to have the Docker client tool installed (to communicate with the docker daemon running on an OpenShift server). If you are using the CDK, the **docker** client tool can conveniently be installed as follows:

```
$ vagrant service-manager install-cli docker
```

For more details, see Preparing Host System for Using Docker from the Command Line in the CDK 2.3 *Getting Started Guide*.

Important

Make sure that you install versions of the **oc** tool and the **docker** tool that are compatible with the version of OpenShift running on the OpenShift Server. An advantage of using the Vagrant **service-manager** approach is that you automatically get the right version of the tools.

3.2. PREPARE YOUR DEVELOPMENT ENVIRONMENT

After installing the required software and tools, prepare your development environment as follows.

3.2.1. Configure Maven Repositories

Configure the Maven repositories, which hold the archetypes and artifacts you will need for building an FIS project on your local machine. Edit your Maven **settings.xml** file, which is usually located in **~/.m2/settings.xml** (on Linux or Max OS) or **Documents and Settings\<USER_NAME>\.m2\settings.xml** (on Windows). The following Maven repositories are required:

- Maven central: **https://repo1.maven.org/maven2**
- Red Hat GA repository: **https://maven.repository.redhat.com/ga**
- Red Hat EA repository: **https://maven.repository.redhat.com/earlyaccess/all**

You must add the preceding repositories both to the dependency repositories section as well as the plug-in repositories section of your **settings.xml** file.

3.2.2. *(Optional)* Install Developer Studio Integration Stack

Red Hat JBoss Developer Studio together with *Developer Studio Integration Stack* is an Eclipse-based development environment, which includes support for developing Fuse Integration Services applications. For details of how to install this development environment, see Install Red Hat JBoss Development Tools from the JBoss Fuse 6.3 *Installation on Apache Karaf* guide.

3.3. (*CDK ONLY*) PREPARE THE VIRTUAL OPENSHIFT SERVER

If you are using the CDK, you need to install the requisite FIS components into the OpenShift Server, as follows:

1. Start the virtual OpenShift Server. Open a command prompt and change to the directory where your **Vagrantfile** is located (typically in the **rhel-ose** subdirectory of the downloaded container tools) before booting the OpenShift image:

```
$ cd <CDK_UNZIPPED>/cdk/components/rhel/rhel-ose
$ vagrant up
```

Note

After the virtual machine has booted, the **vagrant** script prompts you to register the box with **vagrant-registration**. The first time you run the **vagrant** script you must enter the credentials for your Red Hat subscription (the same username and password you would normally use to log on to the Customer Portal, https://access.redhat.com).

2. Log in to the virtual OpenShift Server as an administrator, as follows:

```
oc login -u admin -p admin
```

Note

The **admin** user is a standard account that is automatically created on the virtual OpenShift Server by the CDK.

3. Install the FIS image streams. Enter the following commands at a command prompt:

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-
templates/GA
oc replace --force -n openshift -f ${BASEURL}/fis-image-
streams.json
```

4. Install the FIS templates. Enter the following commands at a command prompt:

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-
templates/GA
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-
camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-
camel-log-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-
camel-rest-sql-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-
cxf-rest-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-
boot-camel-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-
boot-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-
boot-camel-config-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-
boot-camel-drools-template.json
```

```
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-infinispan-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-rest-sql-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-teiid-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-xml-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-cxf-jaxrs-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-cxf-jaxws-template.json
```

3.4. CREATE AND DEPLOY A PROJECT USING THE S2I BINARY WORKFLOW

In this section, you will use the OpenShift S2I binary workflow to create, build, and deploy an FIS project.

1. Create a new FIS project using a Maven archetype. For this example, we use an archetype that creates a sample Spring Boot Camel project. Open a new command prompt and enter the following Maven command:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
  -DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-catalog/2.2.195.redhat-000004/archetypes-catalog-2.2.195.redhat-000004-archetype-catalog.xml \
  -DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
  -DarchetypeArtifactId=spring-boot-camel-xml-archetype \
  -DarchetypeVersion=2.2.195.redhat-000004
```

The archetype plug-in switches to interactive mode to prompt you for the remaining fields:

```
Define value for property 'groupId': : org.example.fis
Define value for property 'artifactId': : fis-spring-boot
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.example.fis: :
[INFO] Using property: spring-boot-version = 1.4.1.RELEASE
Confirm properties configuration:
groupId: org.example.fis
artifactId: fis-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
spring-boot-version: 1.4.1.RELEASE
 Y: :
```

When prompted, enter **`org.example.fis`** for the **`groupId`** value and **`fis-spring-boot`** for the **`artifactId`** value. Accept the defaults for the remaining fields.

2. If the previous command exited with the **`BUILD SUCCESS`** status, you should now have a

new FIS project under the **fis-spring-boot** subdirectory. You can inspect the XML DSL code in the **fis-spring-boot/src/main/resources/spring/camel-context.xml** file. The demonstration code defines a simple Camel route that continuously sends message containing a random number to the log.

3. In preparation for building and deploying the FIS project, log in to the OpenShift Server as follows:

```
oc login -u openshift-dev -p devel https://10.1.2.2:8443
```

> **Note**
>
> The **openshift-dev** user (with **devel** password) is a standard account that is automatically created on the virtual OpenShift Server by the CDK. If you are accessing a remote server, use the URL and credentials provided by your OpenShift administrator.

4. Create a new project namespace called **test** (assuming it does not already exist), as follows:

```
oc new-project test
```

If the **test** project namespace already exists, you can switch to it using the following command:

```
oc project test
```

5. You are now ready to build and deploy the **fis-spring-boot** project. Assuming you are still logged into OpenShift, change to the directory of the **fis-spring-boot** project, and then build and deploy the project, as follows:

```
cd fis-spring-boot
mvn fabric8:deploy
```

At the end of a successful build, you should see some output like the following:

```
...
[INFO] OpenShift platform detected
[INFO] Using project: test
[INFO] Creating a Service from openshift.yml namespace test name
fis-spring-boot
[INFO] Created Service: target/fabric8/applyJson/test/service-
fis-spring-boot.json
[INFO] Updating ImageStream fis-spring-boot from openshift.yml
[INFO] Creating a DeploymentConfig from openshift.yml namespace
test name fis-spring-boot
[INFO] Created DeploymentConfig:
target/fabric8/applyJson/test/deploymentconfig-fis-spring-
boot.json
[INFO] F8: HINT: Use the command `oc get pods -w` to watch your
pods start up
[INFO] ------------------------------------------------------------
-----------------
```

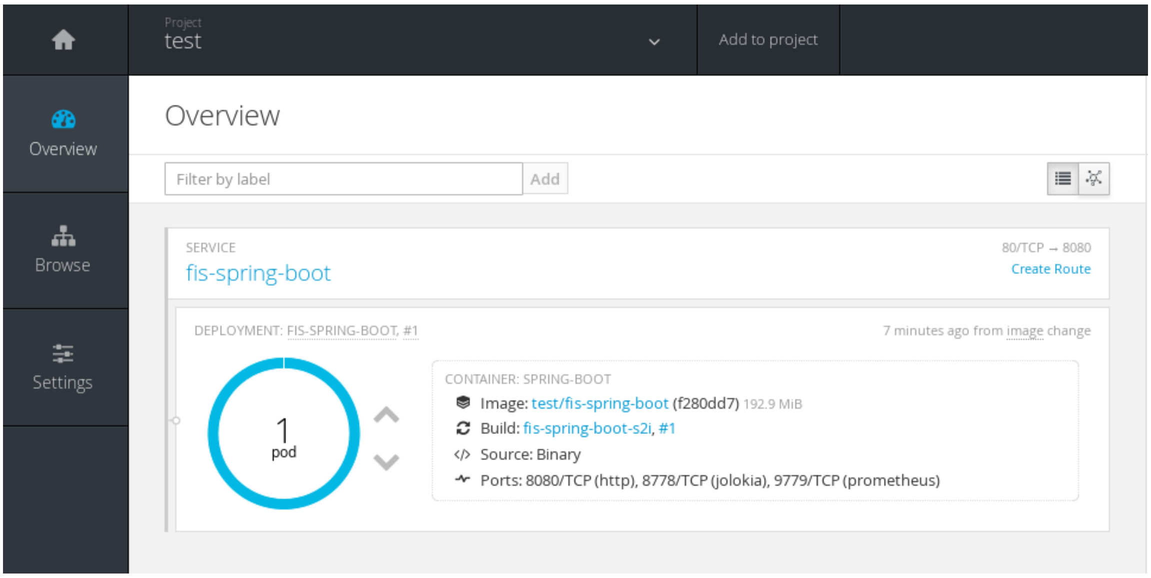
```
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------------------
-----------------
[INFO] Total time: 01:23 min
[INFO] Finished at: 2016-11-10T17:46:05+01:00
[INFO] Final Memory: 66M/666M
[INFO] -------------------------------------------------------
-----------------
```

Note

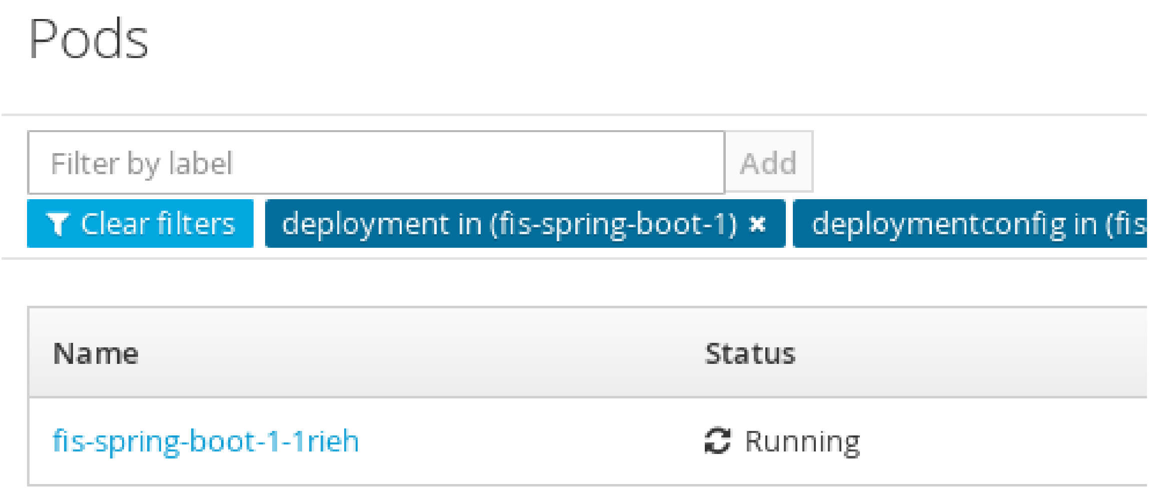
The first time you run this command, Maven has to download a lot of dependencies, which takes several minutes. Subsequent builds will be faster.

6. Navigate to the OpenShift console in your browser (https://10.1.2.2:8443, in the case of the CDK) and log in to the console with your credentials (for example, with username **`openshift-dev`** and password, **`devel`**).

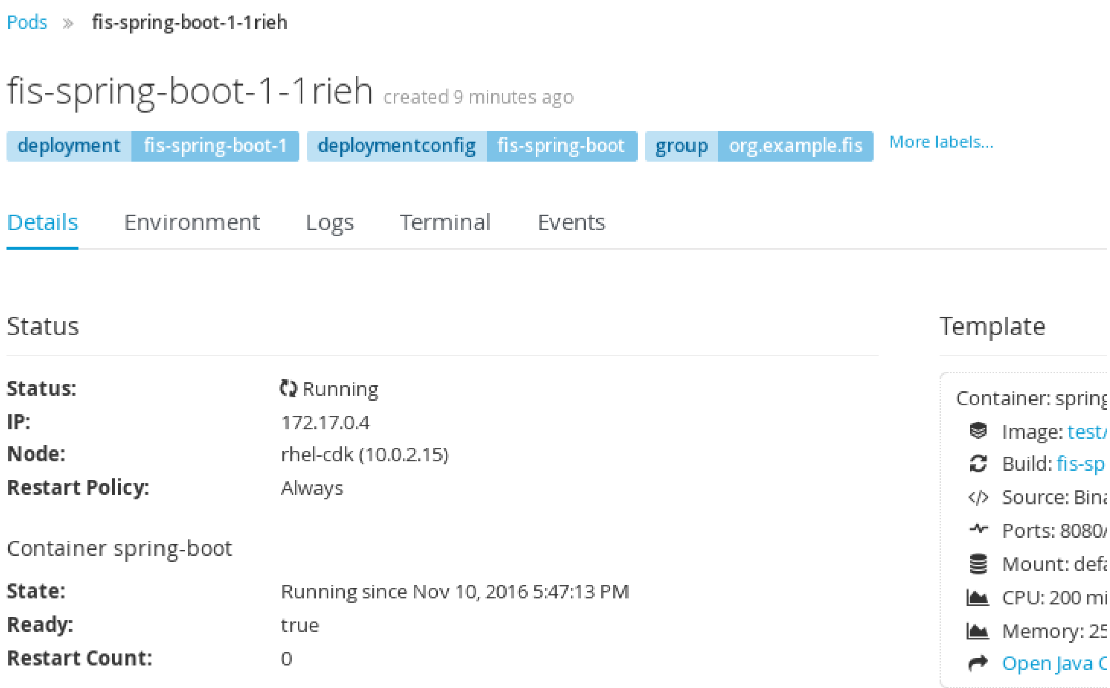
7. In the OpenShift console, scroll down to find the **test** project namespace. Click the **test** project and an overview of the **fis-spring-boot** service opens, as shown.



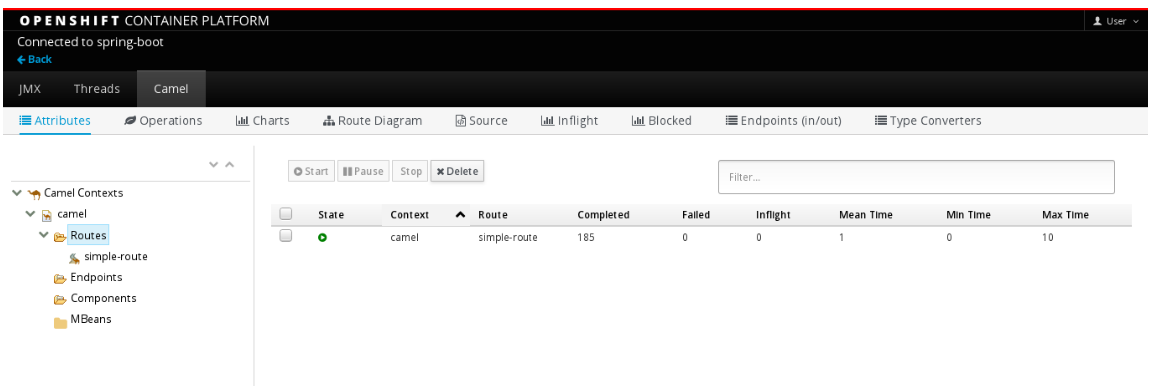
8. Click in the centre of the pod icon (blue circle) to view the list of pods for **fis-spring-boot**.



9. Click on the pod **Name** (in this example, **`fis-spring-boot-1-1rieh`**) to view the details of the running pod.



10. From this view, click on the **Open Java Console** link to open the **HawtIO** console



Note

In order to instruct openshift to display a link to HawtIO console in the pod view, the pod running a FIS image must declare a tcp port within a name attribute set to **`jolokia`**:

```
{
  "kind": "Pod",
  [...]
  "spec": {
    "containers": [
      {
        [...]
        "ports": [
          {
            "name": "jolokia",
            "containerPort": 8778,
            "protocol": "TCP"
          }
```

11. Click on the **Logs** tab to view the application log and scroll down the log to find the random number log messages generated by the Camel application.

```
...
07:30:32.406 [Camel (camel) thread #0 - timer://foo] INFO
simple-route - >>> 985
07:30:34.405 [Camel (camel) thread #0 - timer://foo] INFO
simple-route - >>> 741
07:30:36.409 [Camel (camel) thread #0 - timer://foo] INFO
simple-route - >>> 796
07:30:38.409 [Camel (camel) thread #0 - timer://foo] INFO
simple-route - >>> 211
07:30:40.411 [Camel (camel) thread #0 - timer://foo] INFO
simple-route - >>> 511
07:30:42.411 [Camel (camel) thread #0 - timer://foo] INFO simple-
route - >>> 942
```

12. Click **Overview** on the left-hand navigation bar to return to the overview of the services in the **`test`** namespace. To shut down the running pod, click the down arrow beside the pod icon. When a dialog prompts you with the question **Scale down deployment fis-spring-boot-1?**, click **Scale Down**.

13. *(Optional)* If you are using the CDK, you can shut down the virtual OpenShift Server completely by returning to the command prompt that you used to start up the virtual machine (that is, in the same directory as the **`Vagrantfile`** file) and entering the following command:

```
vagrant halt
```

3.4.1. Redeploy and Undeploy the Project

To redeploy or undeploy your projects, you can use the following commands.

- To redeploy the project,rerun **`mvn fabric8:deploy`**.
- To undeploy the project, run **`mvn fabric8:undeploy`**.

3.5. CREATE AND DEPLOY A PROJECT USING THE S2I SOURCE WORKFLOW

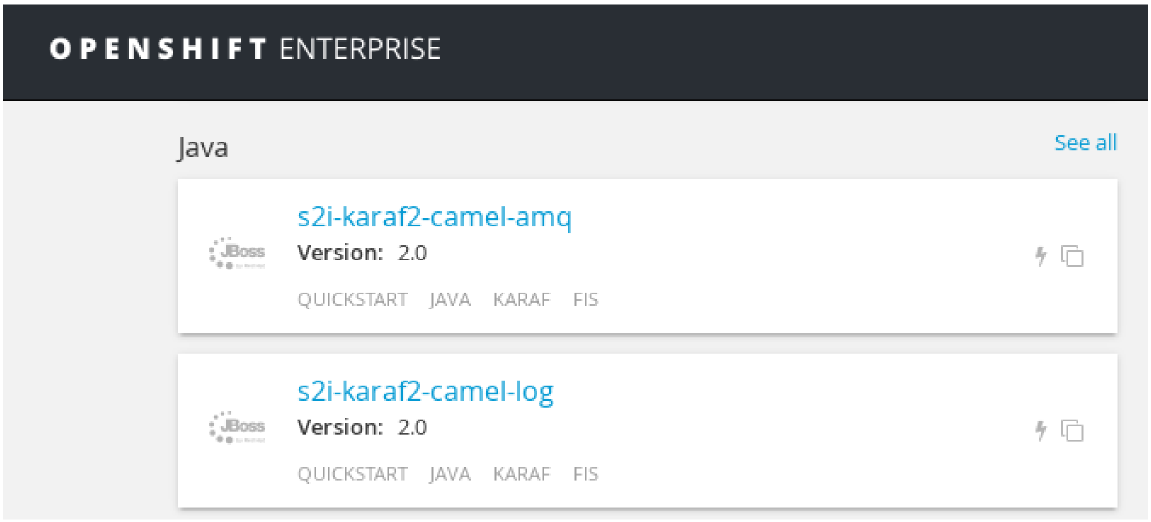
In this section, you will use the OpenShift S2I source workflow to build and deploy an FIS project based on a template. The starting point for this demonstration is a quickstart project stored in a remote Git repository. Using the OpenShift console, you will download, build, and deploy this quickstart project in the OpenShift server.

1. Navigate to the OpenShift console in your browser (https://10.1.2.2:8443, in the case of the CDK) and log in to the console with your credentials (for example, with username **`openshift-dev`** and password, **`devel`**).
2. On the **Projects** screen of the OpenShift console, click **New Project**.
3. On the **New Project** screen, enter **`test`** in the **Name** field and click **Create**. The **Select Image or Template** screen now opens.

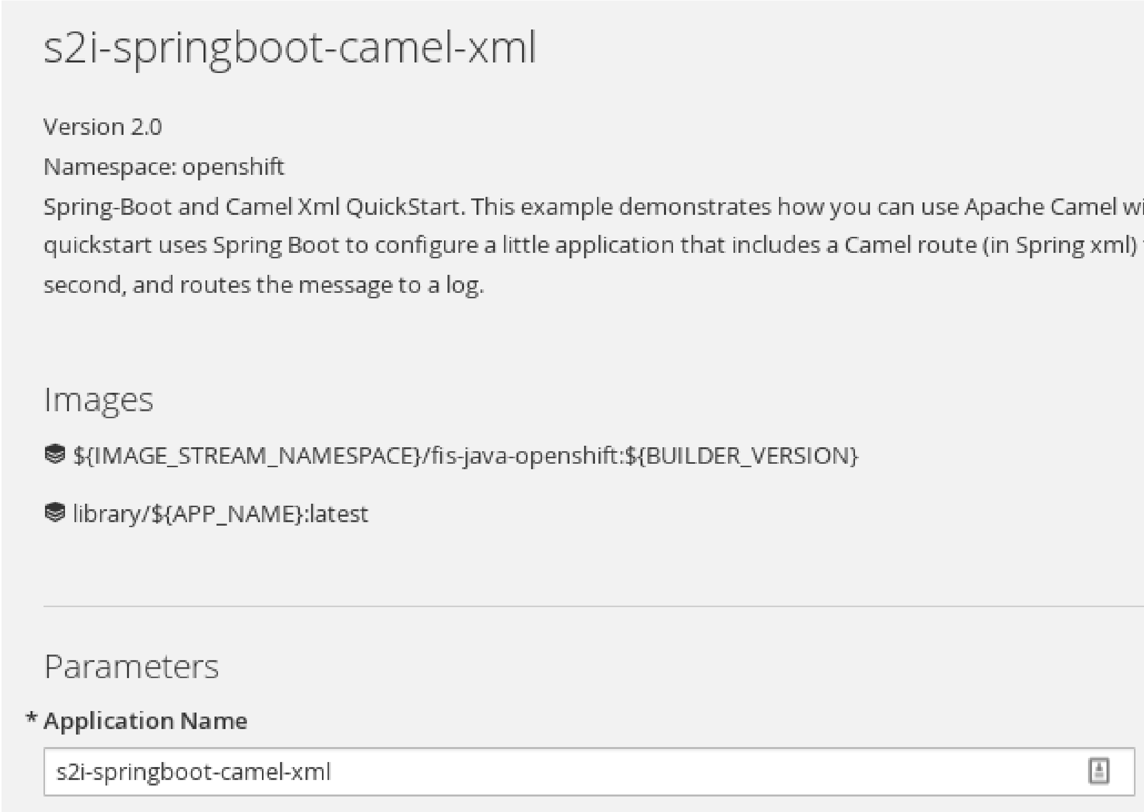
   Note

   If the **`test`** project already exists in OpenShift, click on the **`test`** project and then click **Add to Project** to open the **Select Image or Template** screen.

4. On the **Select Image or Template** screen, scroll down to the **Java** templates and click **See all** to display the full list of Java templates. Now select the **s2i-springboot-camel-xml** template.

5. The **s2i-springboot-camel-xml** template form opens, as shown below. You can accept all of the default settings on this form. Scroll down to the bottom of the form and click **Create**.



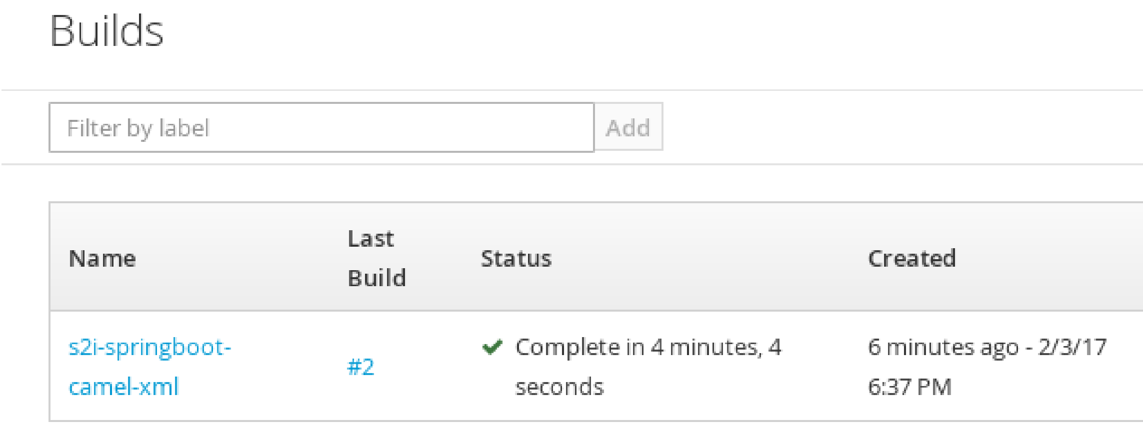
Note

If you want to modify the application code (instead of just running the quickstart as is), you would need to fork the original quickstart Git repository and fill in the appropriate values in the **Git repository URL** and **Git reference** fields.

6. The **Application created** screen now opens. Click **Continue to overview** to go to the **Overview** tab of the OpenShift console (which shows an overview of the available services and pods in the current project). If you have not previously created any application builds in this project, this screen will be empty.

7. In the navigation pane on the left-hand side, select **Browse**→**Builds** to open the **Builds** screen.

8. Click the **`s2i-springboot-camel-xml`** build name to open the **`s2i-springboot-camel-xml`** build page, as shown below.



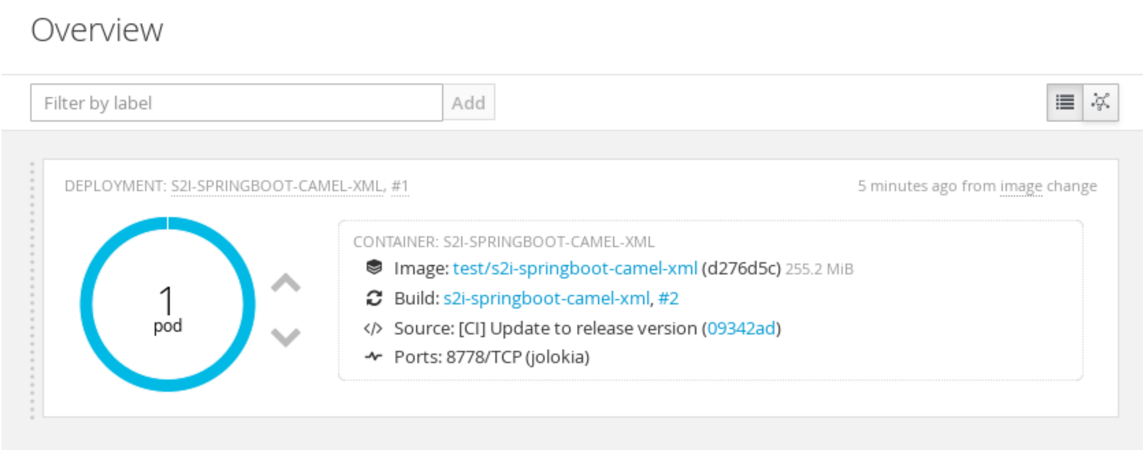
9. Click **View log** to view the log for the latest build — if the build should fail for any reason, the build log can help you to diagnose the problem.

Note

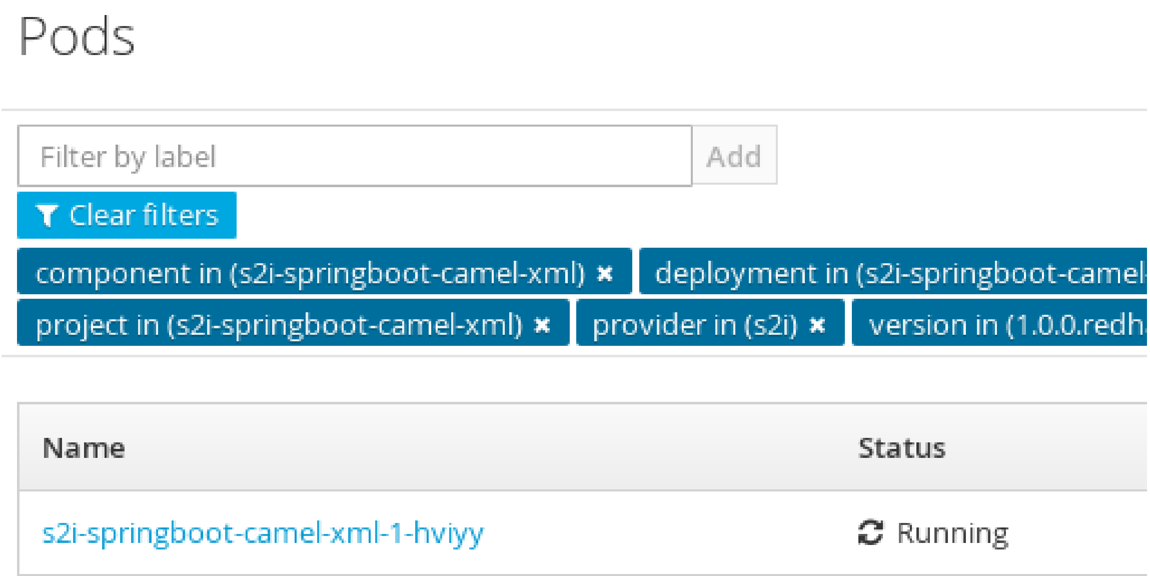
The build can take several minutes to complete, because a lot of dependencies must be downloaded from remote Maven repositories. To speed up build times, we recommend you deploy a Nexus server on your local network.

10. If the build completes successfully, click **Overview** in the left-hand navigation pane to view the running pod for this application.

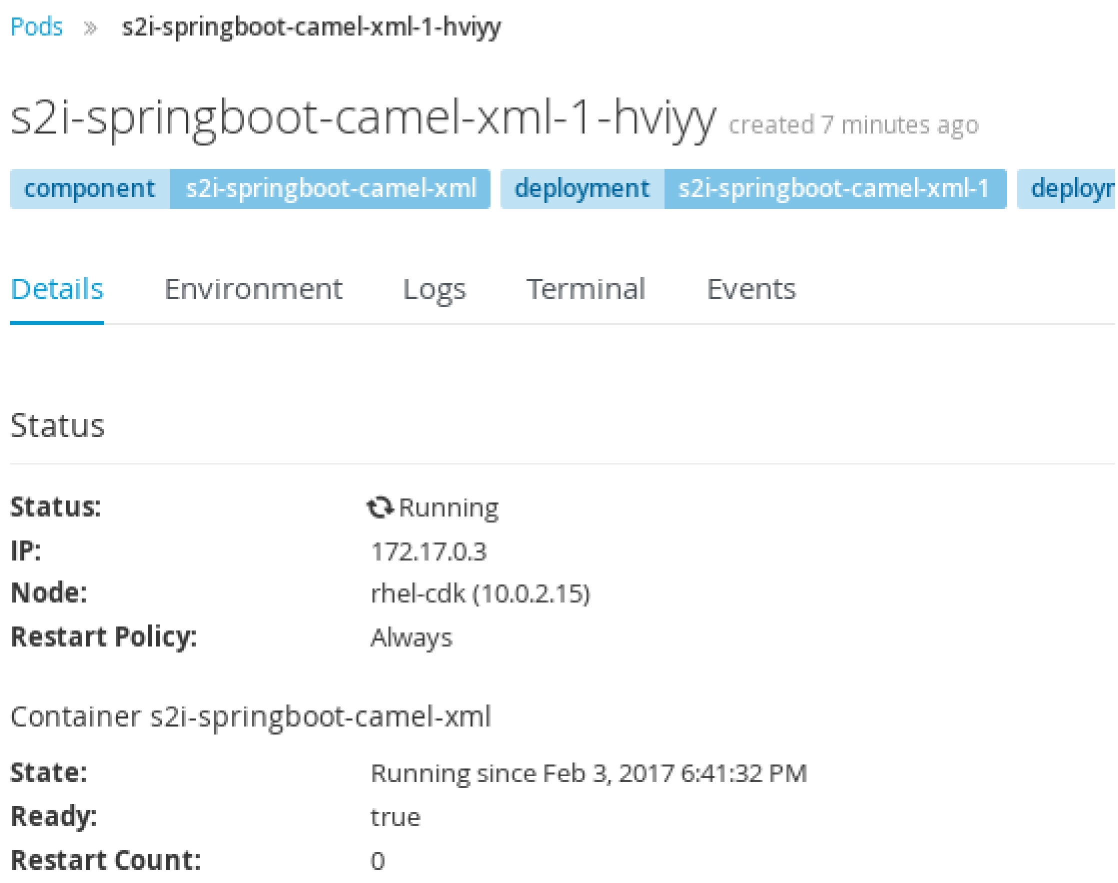
11. Click in the centre of the pod icon (blue circle) to view the list of pods for **s2i-springboot-camel-xml**.



12. Click on the pod **Name** (in this example, **`s2i-springboot-camel-xml-1-hviyy`**) to view the details of the running pod.



13. Click on the **Logs** tab to view the application log and scroll down the log to find the log messages generated by the Camel application.



14. Click **Overview** on the left-hand navigation bar to return to the overview of the services in the `test` namespace. To shut down the running pod, click the down arrow  beside the pod icon. When a dialog prompts you with the question **Scale down deployment s2i-springboot-camel-xml-1?**, click **Scale Down**.

15. *(Optional)* If you are using the CDK, you can shut down the virtual OpenShift Server completely by returning to the command prompt that you used to start up the virtual machine (that is, in the same directory as the **`Vagrantfile`** file) and entering the following command:

    ```
    vagrant halt
    ```

3.6. TIPS AND TRICKS

Here are a few tips that you might find useful when working with OpenShift.

3.6.1. (*CDK Only*) Using SSH to Log On to the Virtual OpenShift Server

You can log in directly to the RHEL 7 image on the virtual machine using SSH, which gives you a command prompt on the virtual RHEL 7 operating system. Change directory to the location of your **`Vagrantfile`** and use Vagrant to log in, as follows:

```
$ cd <CDK_UNZIPPED>/cdk/components/rhel/rhel-ose
$ vagrant ssh
[vagrant@rhel-cdk ~]$ pwd
/home/vagrant
```

You are not prompted for credentials (the login authentication is based on SSH keys that Vagrant has automatically configured and installed). You will find that you are logged in as the user, **`vagrant`**. If you need superuser permissions on the virtual machine, use **`sudo`**. The **`vagrant`** user has **`sudo`** permissions and you are not required to enter a password when you invoke **`sudo`**.

3.6.2. (*CDK Only*) Sharing Files with the Virtual Machine

Vagrant automatically mounts your home directory on the virtual machine, which is convenient for transferring data and sharing files. For example, if your username is **`myusername`**, after you log in to the virtual machine through SSH you will be able see the contents of your home directory:

```
$ cd <CDK_UNZIPPED>/cdk/components/rhel/rhel-ose
$ vagrant ssh
[vagrant@rhel-cdk ~]$ ls /home
myusername  vagrant
[vagrant@rhel-cdk ~]$ cd /home/myusername
```

This feature is implemented by the CDK's **`sshfs`** plug-in for Vagrant.

3.6.3. (*CDK Only*) Customizing the Docker Daemon Configuration

If you want to customize the configuration of the Docker daemon running on the virtual machine, log into the virtual machine and edit the Docker daemon configuration file, as follows:

1. Log in to the virtual machine:

    ```
    $ cd <CDK_UNZIPPED>/cdk/components/rhel/rhel-ose
    $ vagrant ssh
    [vagrant@rhel-cdk ~]$
    ```

2. Edit the Docker daemon configuration file:

```
[vagrant@rhel-cdk ~]$ sudo vi /etc/sysconfig/docker
```

3. Restart the Docker daemon:

```
[vagrant@rhel-cdk ~]$ sudo systemctl restart docker.service
```

CHAPTER 4. DEVELOP AN APPLICATION FOR THE SPRING BOOT IMAGE

4.1. OVERVIEW

This chapter explains how to develop applications for the Spring Boot image.

4.2. CREATE A SPRING BOOT PROJECT USING MAVEN ARCHETYPE

To create a Spring Boot project, follow these steps:

1. Go to the appropriate directory on your system.

2. Launch the **mvn** command to create Spring Boot project

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate
\
  -
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabri
c8/archetypes/archetypes-catalog/2.2.195.redhat-
000004/archetypes-catalog-2.2.195.redhat-000004-archetype-
catalog.xml \
  -DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
  -DarchetypeArtifactId=spring-boot-camel-xml-archetype \
  -DarchetypeVersion=2.2.195.redhat-000004
```

The archetype plug-in switches to interactive mode to prompt you for the remaining fields

```
Define value for property 'groupId': : org.example.fis
Define value for property 'artifactId': : fis-spring-boot
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.example.fis: :

[INFO] Using property: spring-boot-version = 1.4.1.RELEASE
Confirm properties configuration:
groupId: org.example.fis
artifactId: fis-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
spring-boot-version: 1.4.1.RELEASE
 Y: :
```

When prompted, enter **org.example.fis** for the **groupId** value and **fis-spring-boot** for the **artifactId** value. Accept the defaults for the remaining fields.

Then, follow the instructions in the quickstart on how to build and deploy the example.

Note

For the full list of available Spring Boot archetypes, see Section 4.4, “Spring Boot Archetype Catalog”.

4.3. STRUCTURE OF THE CAMEL SPRING BOOT APPLICATION

The directory structure of a Camel Spring Boot application is as follows:

```
├── LICENSE.md
├── pom.xml
├── README.md
└── src
    ├── main
    │   ├── fabric8
    │   │   └── deployment.yml
    │   ├── java
    │   │   └── org
    │   │       └── first1
    │   │           └── spring
    │   │               └── boot
    │   │                   └── project
    │   │                       ├── Application.java
    │   │                       └── MyTransformer.java
    │   └── resources
    │       ├── application.properties
    │       ├── logback.xml
    │       └── spring
    │           └── camel-context.xml
    └── test
        ├── java
        │   └── org
        │       └── first1
        │           └── spring
        │               └── boot
        │                   └── project
        │                       └── KubernetesIntegrationKT.java
        └── resources
```

Where the following files are important for developing an application:

pom.xml

Includes additional dependencies. Camel components that are compatible with Spring Boot are available in the starter version, for example **`camel-jdbc-starter`** or **`camel-infinispan-starter`**. Once the starters are included in the **`pom.xml`** they are automatically configured and registered with the Camel content at boot time. Users can configure the properties of the components using the **`application.properties`** file.

application.properties

It is an important file that allows you to externalize your configuration and work with the same application code in different environments. For details, see Externalized Configuration

For example, in this Camel application you can configure certain properties such as name of the application or the IP addresses, and so on.

application.properties

```
# the options from
org.apache.camel.spring.boot.CamelConfigurationProperties can be
configured here
camel.springboot.name=MyCamel

# lets listen on all ports to ensure we can be invoked from the
pod IP
server.address=0.0.0.0
management.address=0.0.0.0
```

Application.java

It is an important file to run your application. As a user you will import here a file **`camel-context.xml`** to configure routes using the Spring DSL.

The **`Application.java file`** specifies the **`@SpringBootApplication`** annotation, which is equivalent to **`@Configuration`**, **`@EnableAutoConfiguration`** and **`@ComponentScan`** with their default attributes.

Application.java

```
@SpringBootApplication
// load regular Spring XML file from the classpath that contains the
Camel XML DSL
@ImportResource({"classpath:spring/camel-context.xml"})
```

It must have a **`main`** method to run the Spring Boot application.

Application.java

```
public class Application {
    /**
     * A main method to start this application.
     */
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

camel-context.xml

The **`src/main/resources/spring/camel-context.xml`** is an important file for developing application as it contains the Camel routes.

Note

You can find more information on developing Spring-Boot applications at Developing your first Spring Boot Application

src/main/fabric8/deployment.yml

Provides additional configuration that is merged with the default OpenShift configuration file generated by the fabric8-maven-plugin.

Note

This file is not used part of Spring Boot application but it is used in all quickstarts to limit the resources such as CPU and memory usage.

KubernetesIntegrationKT.java

An Arquillian based integration test that test deploying into OpenShift and making sure the container can boot correctly.

4.4. SPRING BOOT ARCHETYPE CATALOG

The Spring Boot Archetype catalog includes the following examples.

Table 4.1. Spring Boot Maven Archetypes

Name	Description
`spring-boot-camel-archetype`	Demonstrates how to use Apache Camel with Spring Boot based on a fabric8 Java base image.
`spring-boot-camel-amq-archetype`	Demonstrates how to connect a Spring-Boot application to an ActiveMQ broker and use JMS messaging between two Camel routes using Kubernetes or OpenShift.
`spring-boot-camel-config-archetype`	Demonstrates how to configure a Spring-Boot application using Kubernetes ConfigMaps and Secrets.
`spring-boot-camel-drools-archetype`	Demonstrates how to use Apache Camel to integrate a Spring-Boot application running on Kubernetes or OpenShift with a remote Kie Server.

Name	Description
`spring-boot-camel-infinispan-archetype`	Demonstrates how to connect a Spring-Boot application to a JBoss Data Grid or Infinispan server using the Hot Rod protocol.
`spring-boot-camel-rest-sql-archetype`	Demonstrates how to use SQL via JDBC along with Camel's REST DSL to expose a RESTful API.
`spring-boot-camel-teiid-archetype`	Demonstrates how to connect Apache Camel to a remote JBoss Data Virtualization (or Teiid) Server using the JDBC protocol.
`spring-boot-camel-xml-archetype`	Demonstrates how to configure Camel routes in Spring Boot via a Spring XML configuration file.
`spring-boot-cxf-jaxrs-archetype`	Demonstrates how to use Apache CXF with Spring Boot based on a fabric8 Java base image. The quickstart uses Spring Boot to configure an application that includes a CXF JAXRS endpoint with Swagger enabled.
`spring-boot-cxf-jaxws-archetype`	Demonstrates how to use Apache CXF with Spring Bootbased on a fabric8 Java base image. The quickstart uses Spring Boot to configure an application that includes a CXF JAXWS endpoint.

4.5. CAMEL STARTER MODULES

4.5.1. Overview

Starters are Apache Camel modules intended to be used in Spring Boot applications. There is a **`camel-xxx-starter`** module for each Camel component (with few exceptions listed below).

Starters meet the following requirements:

- Allow auto-configuration of the component using native Spring Boot configuration system which is compatible with IDE tooling.
- Allows auto-configuration of data formats and languages.
- Manage transitive logging dependencies to integrate with Spring Boot logging system.
- Include additional dependencies and align transitive dependencies to minimize the effort of creating a working Spring Boot application.

Each starter has its own integration test in **`tests/camel-itest-spring-boot`**, that verifies the compatibility with the current release of Spring Boot.

4.5.2. Using Camel Starter Modules

Apache Camel provides a starter module that allows you to develop Spring Boot applications using starters.

To use the Spring Boot starter:

1. Add the following to your Spring Boot pom.xml file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-boot-starter</artifactId>
</dependency>
```

1. Add classes with your Camel routes such as:

```
package com.example;

import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;

@Component
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("timer:foo")
          .to("log:bar");
    }
}
```

These routes will be started automatically.

Note

To keep the main thread blocked so that Camel stays up, either include the **`spring-boot-starter-web`** dependency, or add **`camel.springboot.main-run-controller=true`** to your **`application.properties`** or **`application.yml`** file.

You can customize the Camel application in the **`application.properties`** or **`application.yml`** file with **`camel.springboot.* properties`**.

4.6. UNSUPPORTED STARTER MODULES

The following components do not have a starter because of compatibility issues:

- **camel-blueprint** (intended for OSGi only)
- **camel-cdi** (intended for CDI only)

- **camel-core-osgi** (intended for OSGi only)
- **camel-ejb** (intended for JEE only)
- **camel-eventadmin** (intended for OSGi only)
- **camel-ibatis** (`camel-mybatis-starter` is included)
- **camel-jclouds**
- **camel-mina** (`camel-mina2-starter` is included)
- **camel-paxlogging** (intended for OSGi only)
- **camel-quartz** (`camel-quartz2-starter` is included)
- **camel-spark-rest**
- **camel-swagger** (`camel-swagger-java-starter` is included)

CHAPTER 5. DEVELOP CAMEL APPLICATIONS IN SPRING BOOT

5.1. INTRODUCTION TO CAMEL SPRING BOOT

Spring Boot component provides auto configuration for Apache Camel. Auto-configuration of the Camel context auto-detects Camel routes available in the Spring context and registers the key Camel utilities such as producer template, consumer template, and the type converter as beans.

Every Camel Spring Boot application should use **dependencyManagement** with productized versions, see quickstart pom. Versions that are tagged later can be omitted to not override the versions from BOM.

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>io.fabric8</groupId>
   <artifactId>fabric8-project-bom-camel-spring-boot</artifactId>
   <version>${fabric8.version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
</dependencyManagement>
```

Note

camel-spring-boot jar comes with the **spring.factories** file which allows you to add that dependency into your classpath and hence Spring Boot will automatically auto-configure Camel.

5.2. INTRODUCTION TO CAMEL SPRING BOOT STARTER

Apache Camel contains Spring Boot Starter module that allows you to develop Spring Boot applications using starters.

Note

For more details, see sample application in the source code.

To use the starter, add the following snippet to your Spring Boot **pom.xml** file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-boot-starter</artifactId>
</dependency>
```

The starter allows you to add classes with your Camel routes, as shown in the snippet below. Once these routes are added to the class path the routes are started automatically.

```
package com.example;

import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;

@Component
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("timer:foo").to("log:bar");
    }
}
```

Note

You can customize the Camel application in the **`application.properties`** or **`application.yml`** file.

5.3. AUTO-CONFIGURED CAMEL CONTEXT

Camel auto-configuration provides a **`CamelContext`** instance and creates a **`SpringCamelContext`**. It also initializes and performs shutdown of that context. This Camel context is registered in the Spring application context under **`camelContext`** bean name and you can access it like other Spring bean.

For example, you can access the **`camelContext`** as shown below:

```
@Configuration
public class MyAppConfig {

  @Autowired
  CamelContext camelContext;

  @Bean
  MyService myService() {
    return new DefaultMyService(camelContext);
  }

}
```

5.4. AUTO-DETECTING CAMEL ROUTES

Camel auto configuration collects all the **`RouteBuilder`** instances from the Spring context and automatically injects them into the **`CamelContext`**. It simplifies the process of creating new Camel route with the Spring Boot starter. You can create the routes by adding the **`@Component`** annotated class to your classpath.

```
@Component
public class MyRouter extends RouteBuilder {
```

```
  @Override
  public void configure() throws Exception {
    from("jms:invoices").to("file:/invoices");
  }

}
```

To create a new route **RouteBuilder** bean in your **@Configuration** class, see below:

```
@Configuration
public class MyRouterConfiguration {

  @Bean
  RoutesBuilder myRouter() {
    return new RouteBuilder() {

      @Override
      public void configure() throws Exception {
        from("jms:invoices").to("file:/invoices");
      }

    };
  }

}
```

5.5. CAMEL PROPERTIES

Spring Boot auto configuration automatically connects to Spring Boot external configuration such as properties placeholders, OS environment variables, or system properties with Camel properties support.

These properties are defined in **application.properties** file:

```
route.from = jms:invoices
```

Use as system property

```
java -Droute.to=jms:processed.invoices -jar mySpringApp.jar
```

Use as placeholders in Camel route:

```
@Component
public class MyRouter extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("{{route.from}}").to("{{route.to}}");
  }

}
```

5.6. CUSTOM CAMEL CONTEXT CONFIGURATION

To perform operations on **CamelContext** bean created by Camel auto configuration, you need to register **CamelContextConfiguration** instance in your Spring context as shown below:

```
@Configuration
public class MyAppConfig {

  ...

  @Bean
  CamelContextConfiguration contextConfiguration() {
    return new CamelContextConfiguration() {
      @Override
      void beforeApplicationStart(CamelContext context) {
        // your custom configuration goes here
      }
    };
  }

}
```

> **Note**
>
> The method **CamelContextConfiguration** and **beforeApplicationStart(CamelContext)** will be called before the Spring context is started, so the **CamelContext** instance passed to this callback is fully auto-configured. You can add many instances of **CamelContextConfiguration** into your Spring context and all of them will be executed.

5.7. DISABLING JMX

To disable JMX of the auto-configured **CamelContext** use **camel.springboot.jmxEnabled** property as JMX is enabled by default.

For example, you could add the following property to your **application.properties** file:

```
camel.springboot.jmxEnabled = false
```

5.8. AUTO-CONFIGURED CONSUMER AND PRODUCER TEMPLATES

Camel auto configuration provides pre-configured **ConsumerTemplate** and **ProducerTemplate** instances. You can inject them into your Spring-managed beans:

```
@Component
public class InvoiceProcessor {

  @Autowired
  private ProducerTemplate producerTemplate;

  @Autowired
  private ConsumerTemplate consumerTemplate;
```

```
  public void processNextInvoice() {
    Invoice invoice = consumerTemplate.receiveBody("jms:invoices",
Invoice.class);
    ...
    producerTemplate.sendBody("netty-http:http://invoicing.com/received/"
+ invoice.id());
  }

}
```

By default consumer templates and producer templates come with the endpoint cache sizes set to 1000. You can change those values using the following Spring properties:

```
camel.springboot.consumerTemplateCacheSize = 100
camel.springboot.producerTemplateCacheSize = 200
```

5.9. AUTO-CONFIGURED TYPECONVERTER

Camel auto configuration registers a **TypeConverter** instance named **typeConverter** in the Spring context.

```
@Component
public class InvoiceProcessor {

  @Autowired
  private TypeConverter typeConverter;

  public long parseInvoiceValue(Invoice invoice) {
    String invoiceValue = invoice.grossValue();
    return typeConverter.convertTo(Long.class, invoiceValue);
  }

}
```

5.10. SPRING TYPE CONVERSION API BRIDGE

Spring consist of http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html#core-convert[type conversion API]. Spring API is similar to the Camel type converter API. Due to the similarities between the two APIs Camel Spring Boot automatically registers a bridge converter (**SpringTypeConverter**) that delegates to the Spring conversion API. That means that out-of-the-box Camel will treat Spring Converters similar to Camel.

This allows you to access both Camel and Spring converters using the Camel **TypeConverter** API, as shown below:

```
@Component
public class InvoiceProcessor {

  @Autowired
  private TypeConverter typeConverter;

  public UUID parseInvoiceId(Invoice invoice) {
    // Using Spring's StringToUUIDConverter
```

```
    UUID id = invoice.typeConverter.convertTo(UUID.class,
invoice.getId());
  }

}
```

Here, Spring Boot delegates conversion to the Spring's **ConversionService** instances available in the application context. If no **ConversionService** instance is available, Camel Spring Boot auto configuration creates an instance of **ConversionService**.

5.11. DISABLING TYPE CONVERSIONS FEATURES

To disable registering type conversion features of Camel Spring Boot such as **TypeConverter** instance or Spring bridge, set the **camel.springboot.typeConversion** property to **false** as shown below:

```
camel.springboot.typeConversion = false
```

5.12. ADDING XML ROUTES

By default, you can put Camel XML routes in the classpath under the directory camel, which **camel-spring-boot** will auto detect and include. From **Camel version 2.17** onwards you can configure the directory name or disable this feature using the configuration option, as shown below:

```
// turn off
camel.springboot.xmlRoutes = false
// scan in the com/foo/routes classpath
camel.springboot.xmlRoutes = classpath:com/foo/routes/*.xml
```

Note

The XML files should be Camel XML routes and not **CamelContext** such as:

```
    <routes xmlns="http://camel.apache.org/schema/spring">
         <route id="test">
             <from uri="timer://trigger"/>
             <transform>
                 <simple>ref:myBean</simple>
             </transform>
             <to uri="log:out"/>
         </route>
     </routes>
```

5.13. ADDING XML REST-DSL

By default, you can put Camel Rest-DSL XML routes in the classpath under the directory **camel-rest**, which **camel-spring-boot** will auto detect and include. You can configure the directory name or disable this feature using the configuration option, as shown below:

```
// turn off
camel.springboot.xmlRests = false
// scan in the com/foo/routes classpath
camel.springboot.xmlRests = classpath:com/foo/rests/*.xml
```

Note

The Rest-DSL XML files should be Camel XML rests and not **`CamelContext`** such as:

```
<rests xmlns="http://camel.apache.org/schema/spring">
   <rest>
      <post uri="/persons">
         <to uri="direct:postPersons"/>
      </post>
      <get uri="/persons">
         <to uri="direct:getPersons"/>
      </get>
      <get uri="/persons/{personId}">
          <to uri="direct:getPersionId"/>
      </get>
      <put uri="/persons/{personId}">
          <to uri="direct:putPersionId"/>
      </put>
      <delete uri="/persons/{personId}">
          <to uri="direct:deletePersionId"/>
      </delete>
   </rest>
 </rests>
```

5.14. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 6. INTEGRATE SPRING BOOT WITH KUBERNETES

6.1. INTRODUCTION TO SPRING BOOT WITH KUBERNETES INTEGRATION

6.1.1. What are we Integrating?

The Spring Cloud Kubernetes plug-in currently enables you to integrate the following features of Spring Boot and Kubernetes:

- Section 6.1.2, “Spring Boot Externalized Configuration” integrates with,
- Section 6.1.3, “Kubernetes ConfigMap” and,
- Section 6.1.4, “Kubernetes Secrets”

6.1.2. Spring Boot Externalized Configuration

In Spring Boot, externalized configuration is the mechanism that enables you to inject configuration values from external sources into Java code. In your Java code, injection is typically enabled by annotating with the **`@Value`** annotation (to inject into a single field) or the **`@ConfigurationProperties`** annotation (to inject into multiple properties on a Java bean class).

The configuration data can come from a wide variety of different sources (or *property sources*). In particular, configuration properties are often set in a project’s **`application.properties`** file (or **`application.yaml`** file, if you prefer).

6.1.3. Kubernetes ConfigMap

A Kubernetes ConfigMap is a mechanism that can provide configuration data to a deployed application. A ConfigMap object is typically defined in a YAML file, which is then uploaded to the Kubernetes cluster, making the configuration data available to deployed applications.

6.1.4. Kubernetes Secrets

A Kubernetes Secrets is a mechanism for providing sensitive data (such as passwords, certificates, and so on) to deployed applications.

6.1.5. Spring Cloud Kubernetes Plug-In

The Spring Cloud Kubernetes plug-in implements the integration between Kubernetes and Spring Boot. In principle, you could access the configuration data from a ConfigMap using the Kubernetes API. It is much more convenient, however, to integrate Kubernetes ConfigMap directly with the Spring Boot externalized configuration mechanism, so that Kubernetes ConfigMaps behave as an alternative property source for Spring Boot configuration. This is essentially what the Spring Cloud Kubernetes plug-in provides.

6.1.6. How to Enable Spring Boot with Kubernetes Integration

In a typical Spring Boot Maven project, you can enable the integration by adding the following

Maven dependency to your project's POM file:

```
<project ...>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>io.fabric8</groupId>
      <artifactId>spring-cloud-kubernetes-core</artifactId>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

To complete the integration, you need to add some annotations to your Java source code, create a Kubernetes ConfigMap object, and modify the OpenShift service account permissions to allow your application to read the ConfigMap object. These steps are described in detail in Section 6.2, "Tutorial for ConfigMap Property Source".

6.1.7. Community Demonstration Code

There is some demonstration code available from the Fabric8 community that illustrates the main features of the Spring Cloud Kubernetes plug-in in a simple application. The spring-boot-camel-config demonstration is available from the **`fabric8-quickstarts`** project on GitHub.

Note

The **`spring-boot-camel-config`** demonstration code from the Fabric8 community is *not* supported by Red Hat.

6.2. TUTORIAL FOR CONFIGMAP PROPERTY SOURCE

6.2.1. Tutorial Steps

Perform the following steps to create a simple Camel Spring Boot project, which you can use to experiment with the Kubernetes ConfigMap feature:

1. Create a new Camel Spring Boot project using a Maven archetype. Open a new command prompt and enter the following Maven command:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate
\
  -
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabri
c8/archetypes/archetypes-catalog/2.2.195.redhat-
000004/archetypes-catalog-2.2.195.redhat-000004-archetype-
catalog.xml \
  -DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
  -DarchetypeArtifactId=spring-boot-camel-archetype \
  -DarchetypeVersion=2.2.195.redhat-000004
```

The archetype plug-in switches to interactive mode to prompt you for the remaining fields:

```
Define value for property 'groupId': : org.fis.configmap
Define value for property 'artifactId': : fis-configmap
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.fis.configmap: :
[INFO] Using property: spring-boot-version = 1.4.1.RELEASE
Confirm properties configuration:
groupId: org.fis.configmap
artifactId: fis-configmap
version: 1.0-SNAPSHOT
package: org.fis.configmap
spring-boot-version: 1.4.1.RELEASE
 Y: :
```

When prompted, enter **`org.fis.configmap`** for the **`groupId`** value and **`fis-configmap`** for the **`artifactId`** value. Accept the defaults for the remaining fields.

2. Install the Kubernetes integration plug-in in the **`fis-configmap`** project. In the **`fis-configmap`** project directory, open the **`pom.xml`** file and add the following Maven dependency to the dependencies section of the POM file (but *not* the **`dependencyManagement`** section):

```
<project ...>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>io.fabric8</groupId>
      <artifactId>spring-cloud-kubernetes-core</artifactId>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

This is all you need to do to install the plug-in. Because this artifact is configured as a Spring Boot starter module, it will automatically get installed in the Spring Boot application.

Note

The version of the **`spring-cloud-kubernetes-core`** artifact is already specified in the **`fabric8-project-bom-camel-spring-boot`** BOM file (which is imported by the **`pom.xml`** file). Hence, it is unnecessary to specify the **`version`** of this Maven dependency.

3. Define a Spring Boot configuration bean. In the **`src/main/java/org/fis/configmap`** directory, use a text editor to create the file, **`RouteConfig.java`**, and add the following content:

```
package org.fis.configmap;

import
```

```
org.springframework.boot.context.properties.ConfigurationProperties
;
import org.springframework.context.annotation.Configuration;

@Configuration
@ConfigurationProperties(prefix = "route")
public class RouteConfig {

    protected String logMessage;

    public RouteConfig() {
    }

    public String getLogMessage() {
        return logMessage;
    }

    public void setLogMessage(String logMessage) {
        this.logMessage = logMessage;
    }
}
```

The **`@ConfigurationProperties(prefix = "route")`** annotation associates the **`RouteConfig`** bean properties with the **`route`** property prefix. For example, a Spring Boot property called **`route.logMessage`** or **`route.log-message`** would be injected into the **`logMessage`** bean property.

> **Note**
>
> Kubernetes ConfigMap does not allow you do define property names in CamelCase (it requires property names to be all lowercase). To work around this limitation, use the hyphenated form **`log-message`** in a ConfigMap, which Spring Boot matches with **`logMessage`**. This takes advantage of Spring Boot's relaxed binding rules for externalized configuration.

4. Modify the **`Application.java`** file to read the **`logMessage`** property from the **`RouteConfig`** bean, as follows:

```
...
import org.springframework.beans.factory.annotation.Autowired; (1)

/**
 * A spring-boot application that includes a Camel route builder to
setup the Camel routes
 */
@SpringBootApplication
@ImportResource({"classpath:spring/camel-context.xml"})
public class Application extends RouteBuilder {

    @Autowired
    private RouteConfig config; (2)

    // must have a main method spring-boot can run
```

```
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Override
    public void configure() throws Exception {
        from("timer://foo?period=5000")
            .setBody().method(config, "getLogMessage") 3
            .log(">>> ${body}");
    }
}
```

1

Import the **@Autowired** annotation.

2

Use the **@Autowired** annotation to inject a **RouteConfig** bean into the **config** field.

3

Replace the **constant("Hello World")** message body with an invocation of the **getLogMessage** method, **method(config, "getLogMessage")**.

5. Edit the **application.properties** file in **src/main/resources**, adding the following lines at the end of the file:

```
# User property settings for ConfigMap tutorial
route.logMessage=Hello Universe!
```

6. Create a new file, **bootstrap.properties**, in **src/main/resources**, adding the following lines to the file:

```
# Bootstrap settings for ConfigMap tutorial
spring.application.name=configmap-example
```

The value of **spring.application.name** is crucial to the integration with Kubernetes ConfigMap, because this name establishes the link between the application and its associated ConfigMap. It defaults to **application**.

Note

A **bootstrap.properties** file (or **bootstrap.yml** file) behaves similarly to an **application.properties** file, but it is loaded at an earlier phase of application start-up. It is more reliable to set the properties relating to the Spring Cloud Kubernetes plug-in in the **bootstrap.properties** file.

7. Login to OpenShift and switch to the OpenShift project namespace where you will deploy your application. For example, to login as the **openshift-dev** user and deploy to the **test** project namespace on an OpenShift CDK server, enter the following commands:

```
$ oc login -u openshift-dev -p devel https://10.1.2.2:8443
$ oc project test
```

8. Add the **view** permission to the OpenShift serviceaccount, as follows (assuming you use the **default** serviceaccount for your project):

```
oc policy add-role-to-user view --serviceaccount=default
```

9. Create a YAML file for the config map. Use a text editor to add the following content to the file, **logconfigmap.yaml**:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: configmap-example
data:
  route.log-message: Hello Galaxy!
```

10. Create the Kubernetes ConfigMap in OpenShift, using the following command:

```
oc create -f logconfigmap.yaml
```

Note

You can inspect the newly created ConfigMap, using the command **oc get configmap/configmap-example -o yaml**.

11. Build and deploy the project. From the top-level directory of your project, enter the following command:

```
mvn fabric8:deploy
```

12. View the application log as follows. Open the OpenShift console in your browser and select the relevant project namespace (for example, **test**). Click in the centre of the pod icon for the **fis-configmap** service and then — in the **Pods** view — click on the pod **Name** to view the details of the running pod. Now click on the **Logs** tag to view the application log and scroll down to find the log messages generated by the Camel application.

13. At the bottom of the application log, you should see the message, **Hello Galaxy!**, repeated over and over, which indicates that the ConfigMap has successfully overridden the default message (which was **Hello Universe!**).

You might also like to experiment with the *reload* feature, which makes it possible to reconfigure your application dynamically. To try out this feature, you need to set the relevant reload properties in your project's **bootstrap.properties** file and then try updating the ConfigMap while the application is running (which you can do by editing the **logconfigmap.yaml** file and entering the command, **oc replace -f logconfigmap.yaml**). For details of the reload feature, see Section 6.5, "PropertySource Reload".

6.3. CONFIGMAP PROPERTYSOURCE

Kubernetes has the notion of ConfigMap for passing configuration to the application. The Spring cloud Kubernetes plug-in provides integration with **ConfigMap** to make config maps accessible by Spring Boot.

The **ConfigMap PropertySource** when enabled will look up Kubernetes for a **ConfigMap** named after the application (see **spring.application.name**). If the map is found it will read its data and do the following:

- Section 6.3.1, “Apply Individual Properties”
- Section 6.3.2, “Apply Property Named application.yaml”
- Section 6.3.3, “Apply Property Named application.properties”

6.3.1. Apply Individual Properties

Let’s assume that we have a Spring Boot application named **demo** that uses properties to read its thread pool configuration.

- **pool.size.core**
- **pool.size.maximum**

This can be externalized to config map in YAML format:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  pool.size.core: 1
  pool.size.max: 16
```

6.3.2. Apply Property Named application.yaml

Individual properties work fine for most cases but sometimes we find YAML is more convenient. In this case we use a single property named **application.yaml** and embed our YAML inside it:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yaml: |-
    pool:
      size:
        core: 1
        max:16
```

6.3.3. Apply Property Named application.properties

You can also define the ConfigMap properties in the style of a Spring Boot **`application.properties`** file. In this case we use a single property named **`application.properties`** and list the property settings inside it:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.properties: |-
    pool.size.core: 1
    pool.size.max: 16
```

6.3.4. Deploying a ConfigMap

To deploy a ConfigMap and make it accessible to a Spring Boot application, perform the following steps:

1. In your Spring Boot application, use the externalized configuration mechanism to access the ConfigMap property source. For example, by annotating a Java bean with the **`@Configuration`** annotation, it becomes possible for the bean's property values to be injected by a ConfigMap.
2. In your project's **`bootstrap.properties`** file (or **`bootstrap.yaml`** file), set the **`spring.application.name`** property to match the name of the ConfigMap.
3. Enable the **`view`** permission on the service account that is associated with your application (by default, this would be the service account called **`default`**). For example, to add the **`view`** permission to the **`default`** service account:

   ```
   oc policy add-role-to-user view system:serviceaccount:$(oc
   project -q):default -n $(oc project -q)
   ```

6.4. SECRETS PROPERTYSOURCE

Kubernetes has the notion of Secrets for storing sensitive data such as password, OAuth tokens, etc. The Spring cloud Kubernetes plug-in provides integration with **`Secrets`** to make secrets accessible by Spring Boot.

The **`Secrets`** property source when enabled will look up Kubernetes for **`Secrets`** from the following sources:

1. Reading recursively from secrets mounts
2. Named after the application (see **`spring.application.name`**)
3. Matching some labels

Please note that, by default, consuming Secrets via API (points 2 and 3 above) **is not enabled**.

If the secrets are found, their data is made available to the application.

6.4.1. Example of Setting Secrets

Let’s assume that we have a Spring Boot application named **demo** that uses properties to read its ActiveMQ and PostreSQL configuration.

```
amq.username
amq.password
pg.username
pg.password
```

These secrets can be externalized to **Secrets** in YAML format:

ActiveMQ Secrets

```
apiVersion: v1
kind: Secret
metadata:
  name: activemq-secrets
  labels:
    broker: activemq
type: Opaque
data:
  amq.username: bXl1c2VyCg==
  amq.password: MWYyZDFlMmU2N2Rm
```

PostreSQL Secrets

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secrets
  labels:
    db: postgres
type: Opaque
data:
  amq.username: dXNlcgo=
  amq.password: cGdhZG1pbgo=
```

6.4.2. Consuming the Secrets

You can select the Secrets to consume in a number of ways:

1. By listing the directories where the secrets are mapped:

   ```
   -
   Dspring.cloud.kubernetes.secrets.paths=/etc/secrets/activemq,etc/
   secrets/postgres
   ```

   If you have all the secrets mapped to a common root, you can set them like this:

   ```
   -Dspring.cloud.kubernetes.secrets.paths=/etc/secrets
   ```

2. By setting a named secret:

```
-Dspring.cloud.kubernetes.secrets.name=postgres-secrets
```

3. By defining a list of labels:

```
-Dspring.cloud.kubernetes.secrets.labels.broker=activemq
-Dspring.cloud.kubernetes.secrets.labels.db=postgres
```

6.4.3. Secrets Configuration Properties

You can use the following properties to configure the Secrets property source:

spring.cloud.kubernetes.secrets.enabled
: Enable the Secrets property source. Type is **`Boolean`** and default is **`true`**.

spring.cloud.kubernetes.secrets.name
: Sets the name of the secret to look up. Type is **`String`** and default is **`${spring.application.name}`**.

spring.cloud.kubernetes.secrets.labels
: Sets the labels used to lookup secrets. This property behaves as defined by Map-based binding. Type is **`java.util.Map`** and default is **`null`**.

spring.cloud.kubernetes.secrets.paths
: Sets the paths where secrets are mounted. This property behaves as defined by Collection-based binding. Type is **`java.util.List`** and default is **`null`**.

spring.cloud.kubernetes.secrets.enableApi
: Enable/disable consuming secrets via APIs. Type is **`Boolean`** and default is **`false`**.

Note

Access to secrets via API may be restricted for security reasons — the preferred way is to mount a secret to the POD.

6.5. PROPERTYSOURCE RELOAD

Some applications may need to detect changes on external property sources and update their internal status to reflect the new configuration. The reload feature of Spring Cloud Kubernetes is able to trigger an application reload when a related ConfigMap or Secret change.

This feature is disabled by default and can be enabled using the configuration property **`spring.cloud.kubernetes.reload.enabled=true`** (for example, in the **`bootstrap.properties`** file).

The following levels of reload are supported (property **`spring.cloud.kubernetes.reload.strategy`**):

refresh

(default) only configuration beans annotated with **`@ConfigurationProperties`** or **`@RefreshScope`** are reloaded. This reload level leverages the refresh feature of Spring Cloud Context.

> **Note**
>
> The PropertySource reload feature can only be used for *simple* properties (that is, not collections) when the reload strategy is set to **`refresh`**. Properties backed by collections must not be changed at runtime.

restart_context

the whole Spring *ApplicationContext* is gracefully restarted. Beans are recreated with the new configuration.

shutdown

the Spring *ApplicationContext* is shut down to activate a restart of the container. When using this level, make sure that the lifecycle of all non-daemon threads is bound to the ApplicationContext and that a replication controller or replica set is configured to restart the pod.

6.5.1. Example

Assuming that the reload feature is enabled with default settings (**refresh** mode), the following bean will be refreshed when the config map changes:

```
@Configuration
@ConfigurationProperties(prefix = "bean")
public class MyConfig {

    private String message = "a message that can be changed live";

    // getter and setters

}
```

A way to see that changes effectively happen is creating another bean that prints the message periodically.

```
@Component
public class MyBean {

    @Autowired
    private MyConfig config;

    @Scheduled(fixedDelay = 5000)
    public void hello() {
        System.out.println("The message is: " + config.getMessage());
    }
}
```

The message printed by the application can be changed using a ConfigMap like the following one:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: reload-example
data:
  application.properties: |-
    bean.message=Hello World!
```

Any change to the property named **`bean.message`** in the Config Map associated with the pod will be reflected in the output of the program.

The full example is available in [spring-cloud-kubernetes-reload-example](spring-cloud-kubernetes-examples/spring-cloud-kubernetes-reload-example).

The reload feature supports two operating modes:

event

: *(default)* watches for changes in ConfigMaps or secrets using the Kubernetes API (web socket). Any event will produce a re-check on the configuration and a reload in case of changes. The **`view`** role on the service account is required in order to listen for config map changes. A higher level role (eg. **`edit`**) is required for secrets (secrets are not monitored by default).

polling

: re-creates the configuration periodically from config maps and secrets to see if it has changed. The polling period can be configured using the property **`spring.cloud.kubernetes.reload.period`** and defaults to **15 seconds**. It requires the same role as the monitored property source. This means, for example, that using polling on file mounted secret sources does not require particular privileges.

The following properties can be used to configure the reloading feature:

spring.cloud.kubernetes.reload.enabled

: Enables monitoring of property sources and configuration reload. Type is **`Boolean`** and default is **`false`**.

spring.cloud.kubernetes.reload.monitoring-config-maps

: Allow monitoring changes in config maps. Type is **`Boolean`** and default is **`true`**.

spring.cloud.kubernetes.reload.monitoring-secrets

: Allow monitoring changes in secrets. Type is **`Boolean`** and default is **`false`**.

spring.cloud.kubernetes.reload.strategy

: The strategy to use when firing a reload (**`refresh`**, **`restart_context`**, **`shutdown`**). Type is **`Enum`** and default is **`refresh`**.

spring.cloud.kubernetes.reload.mode

: Specifies how to listen for changes in property sources (**`event`**, **`polling`**). Type is **`Enum`** and default is **`event`**.

spring.cloud.kubernetes.reload.period

The period in milliseconds for verifying changes when using the **polling** strategy. Type is **Long** and default is **15000**.

Note the following points:

- The **spring.cloud.kubernetes.reload.*** properties should not be used in ConfigMaps or Secrets. Changing such properties at run time may lead to unexpected results;
- Deleting a property or the whole config map does not restore the original state of the beans when using the **refresh** level.

CHAPTER 7. DEVELOP AN APPLICATION FOR THE KARAF IMAGE

7.1. CREATE A KARAF PROJECT USING MAVEN ARCHETYPE

To create a Karaf project using a Maven archetype, follow these steps:

1. Go to the appropriate directory on your system.
2. Launch the Maven command to create a Karaf project

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate
\
  -
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabri
c8/archetypes/archetypes-catalog/2.2.195.redhat-
000004/archetypes-catalog-2.2.195.redhat-000004-archetype-
catalog.xml \
  -DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
  -DarchetypeArtifactId=karaf2-camel-amq-archetype \
  -DarchetypeVersion=2.2.195.redhat-000004
```

3. The archetype plug-in switches to interactive mode to prompt you for the remaining fields

```
Define value for property 'groupId': : org.example.fis
Define value for property 'artifactId': : fis-karaf2
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.example.fis: :
```

When prompted, enter **`org.example.fis`** for the **`groupId`** value and **`fis-karaf2`** for the **`artifactId`** value. Accept the defaults for the remaining fields.

Then, follow the instructions in the quickstart on how to build and deploy the example.

Note

For the full list of available Karaf archetypes, see Section 7.3, “Karaf Archetype Catalog”.

7.2. STRUCTURE OF THE CAMEL KARAF APPLICATION

The directory structure of a Camel Karaf application is as follows:

```
  ├── pom.xml
  ├── README.md
  └── src
      ├── main
      │   ├── fabric8
      │   │   └── deployment.yml
      │   ├── java
      │   │   └── org
      │   │       └── first
```

```
        │    │            └── karaf
        │    │                └── project
        │    └── resources
        │        ├── assembly
        │        │   └── etc
        │        │       └── org.ops4j.pax.logging.cfg
        │        └── OSGI-INF
        │            └── blueprint
        │                └── camel-log.xml
        └── test
            ├── java
            │   └── org
            │       └── first
            │           └── karaf
            │               └── project
            └── resources
```

Where the following files are important for developing a Karaf application:

pom.xml

Includes additional dependencies.

You can add dependencies in the **pom.xml** file, for example for logging you can use SLF4J.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
</dependency>
```

org.ops4j.pax.logging.cfg

Demonstrates how to customize log levels, sets logging level to DEBUG instead of the default INFO.

camel-log.xml

Contains the source code of the application.

src/main/fabric8/deployment.yml

Provides additional configuration that is merged with the default OpenShift configuration file generated by the fabric8-maven-plugin.

Note

This file is not used as part of the Karaf application, but it is used in all quickstarts to limit the resources such as CPU and memory usage.

7.3. KARAF ARCHETYPE CATALOG

The Karaf archetype catalog includes the following examples.

Table 7.1. Karaf Maven Archetypes

Name	Description
`karaf2-camel-log-archetype`	Demonstrates a simple Apache Camel application that logs a message to the server log every 5th second.
`karaf2-camel-amq-archetype`	Demonstrates how to use Camel in a Karaf Container using Blueprint to connect to the AMQ xPaaS message broker on OpenShift.
`karaf2-camel-rest-sql-archetype`	Demonstrates how to use SQL via JDBC along with Camel's REST DSL to expose a RESTful API.
`karaf2-cxf-rest-archetype`	Demonstrates how to create a RESTful(JAX-RS) web service using CXF and expose it through the OSGi HTTP Service.

7.4. FABRIC8 KARAF FEATURES

Fabric8 provides support for Apache Karaf making it easier to develop OSGi apps for Kubernetes.

The important features of Fabric8 are as listed below:

- Different strategies to resolve placeholders in Blueprint XML files.
- Environment variables
- System properties
- Services
- Kubernetes ConfigMap
- Kubernetes Secrets
- Using Kubernetes configuration maps to dynamically update the OSGi configuration administration.
- Provides Kubernetes heath checks for OSGi services.

7.4.1. Adding Fabric8 Karaf Features

To use the features, add **`fabric8-karaf-features`** dependency to the project pom file.

```
<dependency>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-karaf-features</artifactId>
  <version>${fabric8.version}</version>
  <classifier>features</classifier>
```

```
  <type>xml</type>
</dependency>
```

These features will be installed into the Karaf server.

7.4.2. Fabric8 Karaf Core Bundle functionalities

The bundle **`fabric8-karaf-core`** provides functionalities used by Blueprint and ConfigAdmin extensions.

To add the feature in a custom Karaf distribution, add it to **`startupFeatures`** in the project **`pom.xml`**

```
<startupFeatures>
  ...
  <feature>fabric8-karaf-core</feature>
  ...
</startupFeatures>
```

7.4.2.1. Property placeholders resolvers

The bundle **`fabric8-karaf-core`** exports a service **`PlaceholderResolver`** with the following interface:

```
public interface PlaceholderResolver {
    /**
     * Resolve a placeholder using the strategy indicated by the prefix
     *
     * @param value the placeholder to resolve
     * @return the resolved value or null if not resolved
     */
    String resolve(String value);

    /**
     * Replaces all the occurrences of variables with their matching
values from the resolver using the given source string as a template.
     *
     * @param source the string to replace in
     * @return the result of the replace operation
     */
    String replace(String value);

    /**
     * Replaces all the occurrences of variables within the given source
builder with their matching values from the resolver.
     *
     * @param value the builder to replace in
     * @rerurn true if altered
     */
    boolean replaceIn(StringBuilder value);

    /**
     * Replaces all the occurrences of variables within the given
dictionary
```

```
     *
     * @param dictionary the dictionary to replace in
     * @rerurn true if altered
     */
    boolean replaceAll(Dictionary<String, Object> dictionary);

    /**
     * Replaces all the occurrences of variables within the given
dictionary
     *
     * @param dictionary the dictionary to replace in
     * @rerurn true if altered
     */
    boolean replaceAll(Map<String, Object> dictionary);
}
```

The **PlaceholderResolver** service acts as a collector for different property placeholder resolution strategies. The resolution strategies it provides by default are listed in the table.

1. List of resolution strategies

Prefix	Example	Description
env	env:JAVA_HOME	lookup the property from OS environment variables.
sys	sys:java.version	lookup the property from Java JVM system properties.
service	service:amq	lookup the property from OS environment variables using the service naming idiom.
service.host	service.host:amq	lookup the property from OS environment variables using the service naming idiom returning the hostname part only.
service.port	service.port:amq	lookup the property from OS environment variables using the service naming idiom returning the port part only.

k8s:map	k8s:map:myMap/myKey	lookup the property from a Kubernetes ConfigMap (via API)
k8s:secret	k8s:secret:amq/password	lookup the property from a Kubernetes Secrets (via API or volume mounts)

The property placeholder service supports the following options:

1. List of property placeholder service options

Name	Default	Description
fabric8.placeholder.prefix	$[	The prefix for the placeholder
fabric8.placeholder.suffix	]	The suffix for the placeholder
fabric8.k8s.secrets.path	null	A comma delimited list of paths were secrets are mapped
fabric8.k8s.secrets.api.enabled	false	Enable/Disable consuming secrets via APIs

To set the property placeholder service options you can use system properties or environment variables or both.

1. To access ConfigMaps on OpenShift the service account needs view permissions

```
oc policy add-role-to-user view system:serviceaccount:$(oc
project -q):default -n $(oc project -q)
```

2. Mount secret to the POD as access to secrets through API might be restricted.

3. Secrets available on the POD as volume mounts are mapped to a directory named as the secret, as shown below

```
containers:
  -
   env:
   - name: FABRIC8_K8S_SECRETS_PATH
     value: /etc/secrets
     volumeMounts:
   - name: activemq-secret-volume
```

```
      mountPath: /etc/secrets/activemq
      readOnly: true
    - name: postgres-secret-volume
      mountPath: /etc/secrets/postgres
      readOnly: true

volumes:
  - name: activemq-secret-volume
  secret:
  secretName: activemq
  - name: postgres-secret-volume
   secret:
  secretName: postgres
```

7.4.2.2. Adding a custom property placeholders resolvers

You can add a custom placeholder resolver to support a specific need, such as custom encryption. You can also use the **PlaceholderResolver** service to make the resolvers available to Blueprint and ConfigAdmin.

To add a custom property placeholders resolvers, follow these steps:

1. Add the following mvn dependency to the project **pom.xml**.

   pom.xml

```
---
<dependency>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-karaf-core</artifactId>
</dependency>
---
```

2. Implement the PropertiesFunction interface and register it as OSGi service using SCR.

```
import
io.fabric8.karaf.core.properties.function.PropertiesFunction;
import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.ConfigurationPolicy;
import org.apache.felix.scr.annotations.Service;

@Component(
    immediate = true,
    policy = ConfigurationPolicy.IGNORE,
    createPid = false
)
@Service(PropertiesFunction.class)
public class MyPropertiesFunction implements PropertiesFunction {
    @Override
    public String getName() {
        return "myResolver";
    }

    @Override
```

```
        public String apply(String remainder) {
            // Parse and resolve remainder
            return remainder;
        }
    }
```

3. You can reference the resolver in Configuration management as follows.

properties

```
my.property = $[myResolver:value-to-resolve]
```

7.4.3. Adding Fabric8 Karaf Config Admin Support

To include Config Admin Support feature in your custom Karaf distribution, add **`fabric8-karaf-cm`** to **`startupFeatures`** in your project **`pom.xml`**

pom.xml

```
<startupFeatures>
  ...
  <feature>fabric8-karaf-cm</feature>
  ...
</startupFeatures>
```

7.4.3.1. Adding ConfigMap injection

The **`fabric8-karaf-cm`** provides a **`ConfigAdmin`** bridge that inject **`ConfigMap`** values in Karaf's **`ConfigAdmin`**.

To be added by the ConfigAdmin bridge, a ConfigMap has to be labeled with **karaf.pid**, where its values corresponds to the pid of your component.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: myconfig
  labels:
    karaf.pid: com.mycompany.bundle
data:
  example.property.1: my property one
  example.property.2: my property two
```

Individual properties work for most cases. But to define your configuration, you can use a single property names. It is same as the pid file in **`karaf/etc`**. For example,

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: myconfig
  labels:
    karaf.pid: com.mycompany.bundle
```

```
data:
  com.mycompany.bundle.cfg: |
    example.property.1: my property one
    example.property.2: my property two
```

7.4.3.2. Configuration plugin

The **`fabric8-karaf-cm`** provides a **`ConfigurationPlugin`** which resolves configuration property placeholders.

To enable property substitution with the **`fabric8-karaf-cm`** plug-in, you must set the Java property, **`fabric8.config.plugin.enabled`** to **`true`**. For example, you can set this property using the **`JAVA_OPTIONS`** environment variable in the Karaf image, as follows:

```
JAVA_OPTIONS=-Dfabric8.config.plugin.enabled=true
```

An example of configuration property placeholders is shown below.

my.service.cfg

```
    amq.usr = $[k8s:secret:$[env:ACTIVEMQ_SERVICE_NAME]/username]
    amq.pwd = $[k8s:secret:$[env:ACTIVEMQ_SERVICE_NAME]/password]
    amq.url = tcp://$[env+service:ACTIVEMQ_SERVICE_NAME]
```

my-service.xml

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

    <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"
               xsi:schemaLocation="
                 http://www.osgi.org/xmlns/blueprint/v1.0.0

https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
                 http://camel.apache.org/schema/blueprint
                 http://camel.apache.org/schema/blueprint/camel-
blueprint.xsd">

      <cm:property-placeholder persistent-id="my.service" id="my.service"
update-strategy="reload"/>

      <bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent">
         <property name="userName"  value="${amq.usr}"/>
         <property name="password"  value="${amq.pwd}"/>
         <property name="brokerURL" value="${amq.url}"/>
      </bean>
    </blueprint>
```

Fabric8 Karaf Config Admin supports the following options.

Name	Default	Description
fabric8.config.plugin.enabled	false	Enable ConfigurationPlugin
fabric8.cm.bridge.enabled	true	Enable ConfigAdmin bridge
fabric8.config.watch	true	Enable watching for ConfigMap changes
fabric8.config.merge	false	Enable merge ConfigMap values in ConfigAdmin
fabric8.config.meta	true	Enable injecting ConfigMap meta in ConfigAdmin bridge
fabric8.pid.label	karaf.pid	Define the label the ConfigAdmin bridge looks for

Important

ConfigurationPlugin requires **Aries Blueprint CM 1.0.9** or above.

7.4.4. Fabric8 Karaf Blueprint Support

The **fabric8-karaf-blueprint** uses Aries PropertyEvaluator and property placeholders resolvers from **fabric8-karaf-core** to resolve placeholders in your Blueprint XML file.

To include the feature for blueprint support in your custom Karaf distribution, add **fabric8-karaf-blueprint** to **startupFeatures** in your project **pom.xml**.

```
<startupFeatures>
  ...
  <feature>fabric8-karaf-blueprint</feature>
  ...
</startupFeatures>
```

The fabric8 evaluator supports chained evaluators, such as **${env+service:MY_ENV_VAR}**. You need to resolve **MY_ENV_VAR** variable against environment variables. The result is then resolved using service function. For example,

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

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

```
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.2.0"
            xsi:schemaLocation="
              http://www.osgi.org/xmlns/blueprint/v1.0.0
              https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
              http://camel.apache.org/schema/blueprint
              http://camel.apache.org/schema/blueprint/camel-blueprint.xsd
              http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.3.0
              http://aries.apache.org/schemas/blueprint-ext/blueprint-ext-
1.3.xsd">

  <ext:property-placeholder evaluator="fabric8" placeholder-prefix="$["
placeholder-suffix="]"/>

  <bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent">
      <property name="userName"
value="$[k8s:secret:$[env:ACTIVEMQ_SERVICE_NAME]/username]"/>
      <property name="password"
value="$[k8s:secret:$[env:ACTIVEMQ_SERVICE_NAME]/password]"/>
      <property name="brokerURL"
value="tcp://$[env+service:ACTIVEMQ_SERVICE_NAME]"/>
  </bean>
</blueprint>
```

Important

Nested property placeholder substitution requires **`Aries Blueprint Core 1.7.0`** or above.

7.4.5. Fabric8 Karaf Health Checks

It is recommended to install the **`fabric8-karaf-checks`** as a startup feature. Once enable, your Karaf server can expose **`http://0.0.0.0:8181/readiness-check`** and **`http://0.0.0.0:8181/health-check`** URLs which can be used by Kubernetes for readiness and liveness probes.

Note

These URLs will only respond with a HTTP 200 status code when the following is true:

- OSGi Framework is started.
- All OSGi bundles are started.
- All boot features are installed.
- All deployed BluePrint bundles are in the created state.
- All deployed SCR bundles are in the active, registered or factory state.
- All web bundles are deployed to the web server.

- All created Camel contexts are in the started state.

You can add the Karaf health checks feature to the project **pom.xml** using **startupFeatures**.

pom.xml

```
<startupFeatures>
  ...
  <feature>fabric8-karaf-checks</feature>
  ...
</startupFeatures>
```

The **fabric8-maven-plugin:resources** goal will detect if your using the **fabric8-karaf-checks** feature and automatically add the Kubernetes for readiness and liveness probes to your container's configuration.

7.4.5.1. Adding Custom Heath Checks

You can provide additional custom heath checks to prevent the Karaf server from receiving user traffic before it is ready to process the requests. TO enable custom health checks you need to implement the **io.fabric8.karaf.checks.HealthChecker** or **io.fabric8.karaf.checks.ReadinessChecker** interfaces and register those objects in the OSGi registry.

Your project will need to add the following mvn dependency to the project **pom.xml** file.

pom.xml

```
<dependency>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-karaf-checks</artifactId>
</dependency>
```

> **Note**
>
> The simplest way to create and registered an object in the OSGi registry is to use SCR.

An example that performs a health check to make sure you have some free disk space, is shown below:

```
import io.fabric8.karaf.checks.*;
import org.apache.felix.scr.annotations.*;
import org.apache.commons.io.FileSystemUtils;
import java.util.Collections;
import java.util.List;

@Component(
    name = "example.DiskChecker",
    immediate = true,
    enabled = true,
    policy = ConfigurationPolicy.IGNORE,
    createPid = false
```

```
)
@Service({HealthChecker.class, ReadinessChecker.class})
public class DiskChecker implements HealthChecker, ReadinessChecker {

    public List<Check> getFailingReadinessChecks() {
        // lets just use the same checks for both readiness and health
        return getFailingHeathChecks();
    }

    public List<Check> getFailingHealthChecks() {
        long free = FileSystemUtils.freeSpaceKb("/");
        if (free < 1024 * 500) {
            return Collections.singletonList(new Check("disk-space-low",
"Only " + free + "kb of disk space left."));
        }
        return Collections.emptyList();
    }
}
```

CHAPTER 8. USING PERSISTENT STORAGE IN FUSE INTEGRATION SERVICES

MavenFuse Integration Services (FIS) applications are based on OpenShift containers, which do not have a persistent filesystem. Every time you start an application it is started in a new container with an immutable Docker image. Hence any persisted data in the file systems is lost when the container stops. But applications need to store some state as data in a persistent store and sometimes applications share access to a common data store. OpenShift platform supports provisioning of external stores as Persistent Storage.

8.1. VOLUMES

OpenShift allows pods and containers to mount Volumes as file systems which are backed by multiple host-local or network attached storage endpoints. Volume types include:

- emptydir (empty directory): This is a default volume type. It is a directory which gets allocated when the pod is created on a local host. It is not copied across the servers and when you delete the pod the directory is removed.
- configmap: It is a directory with contents populated with key-value pairs from a named configmap.
- hostPath (host directory): It is a directory with specific path on any host and it requires elevated privileges.
- secret (mounted secret): Secret volumes mount a named secret to the provided directory.
- persistentvolumeclaim or pvc (persistent volume claim): This links the volume directory in the container to a persistent volume claim you have allocated by name. A persistent volume claim is a request to allocate storage. Note that if your claim is not bound, your pods will not start.

Volumes are configured at the Pod level and can only directly access an external storage using **hostPath**. Hence it is harder to mange the access to shared resources for multiple Pods as **hostPath** volumes.

8.2. PERSISTENTVOLUMES

PersistentVolumes allow cluster administrators to provision cluster wide storage which is backed by various types of network storage like NFS, Ceph RBD, AWS Elastic Block Store (EBS), etc. **PersistentVolumes** also specify capacity, access modes, and recycling policies. This allows pods from multiple Projects to access persistent storage without worrying about the nature of the underlying resource.

See the Configuring Persistent Storage for creating various types of PersistentVolumes.

8.3. SAMPLE PERSISTENTVOLUME CONFIGURATION

The sample configuration below provisions a path on the host machine as a PersistentVolume named pv001:

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
  name: pv0001
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 2Mi
  hostPath:
    path: /data/pv0001/
```

Here the host path is **/data/pv0001** and storage capacity is limited to 2MB. For example, when using OpenShift CDK it will provision the directory **/data/pv0001** from the virtual machine hosting the OpenShift Cluster. To create this **PersistentVolume**, add the above configuration in a file **pv.yaml** and use the command:

```
oc create -f pv.yaml
```

To verify the creation of **PersistentVolume**, use the following command, which will list all the **PersistentVolumes** configured in your OpenShift cluster:

```
oc get pv
```

8.4. PERSISTENTVOLUMECLAIMS

A **PersistentVolume** exposes a storage endpoint as a named entity in an OpenShift cluster. To access this storage from Projects, **PersistentVolumeClaims** must be created that can access the **PersistentVolume**. **PersistentVolumeClaims** are created for each Project with customized claims for a certain amount of storage with certain access modes.

The sample configuration below creates a claim named pvc0001 for 1MB of storage with read-write-once access against a **PersistentVolume** named pv0001.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc0001
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Mi
```

8.5. VOLUMES IN PODS

Pods use Volume Mounts to define the filesystem mount location and Volumes to define reference **PersistentVolumeClaims**. The sample container configuration below mounts **PersistentVolumeClaim** pvc0001 at **/usr/share/data** in its filesystem.

```
spec:
  template:
    spec:
      containers:
        - volumeMounts:
```

```
          - name: vol0001
            mountPath: /usr/share/data
      volumes:
        - name: vol0001
          persistentVolumeClaim:
            claimName: pvc0001
```

Any data written by the application to the directory **/usr/share/data** is now persisted across container restarts. Add this configuration in the file **src/main/fabric8/deployment.yml** in a FIS application and create OpenShift resources using command:

```
mvn fabric8:resource-apply
```

To verify that the created **DeploymentConfiguration** has the volume mount and volume use the command:

```
oc describe deploymentconfig <application-dc-name>
```

For FIS quickstarts, the **<application-dc-name>** is the Maven project name, for example **spring-boot-camel**.

CHAPTER 9. PATCHING FUSE INTEGRATION SERVICES

9.1. PATCHING OVERVIEW

You might need to perform one or more of the following tasks to bring the Fuse Integration Services (FIS) product up to the latest patch level:

Section 9.2, "Patch Application Dependencies"

Update the dependencies in your project POM file, so that your application uses patched versions of the Maven artifacts.

Section 9.3, "Patch the FIS Templates"

Update the FIS templates on your OpenShift server, so that new projects created with the FIS templates use patched versions of the Maven artifacts.

Section 9.4, "Patch the FIS Images"

Update the FIS images on your OpenShift server, so that new application builds are based on patched versions of the Fuse base images.

9.2. PATCH APPLICATION DEPENDENCIES

In the context of Fuse Integration Services (FIS), applications are built entirely using Maven artifacts downloaded from the Red Hat Maven repositories. Hence, to patch your application, all that you need to do is to edit your project's POM file, changing the Maven dependencies to use the appropriate FIS patch version.

It is important to upgrade all of the Maven dependencies for FIS together, so that your project uses dependencies that are all from the same patch version. The FIS project consists of a carefully curated set of Maven artifacts that are built and tested together. If you try to mix and match Maven artifacts from *different* FIS patch levels, you could end up with a configuration that is untested and unsupported by Red Hat. The easiest way to avoid this scenario is to use a Bill of Materials (BOM) file in Maven, which defines the versions of all the Maven artifacts supported by FIS. When you update the version of a BOM file, you automatically update the versions for all the FIS Maven artifacts in your project's POM.

The POM file that is generated by a FIS Maven archetype or by a FIS template in OpenShift has a standard layout that uses a BOM file and defines the versions of certain required plug-ins. It is recommended that you stick to this standard layout in your own applications, because this makes it much easier to patch and upgrade your application's dependencies.

9.2.1. Update Dependencies in a Spring Boot Application

The following code fragment shows the standard layout of a POM file for a Spring Boot application in FIS, highlighting some important property settings:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
    <!-- maven plugin versions -->
    <fabric8.maven.plugin.version>3.1.80.redhat-
000004</fabric8.maven.plugin.version>

    <!-- configure the versions you want to use here -->
    <fabric8.version>2.2.170.redhat-000004</fabric8.version>
    <spring-boot.version>1.4.1.RELEASE</spring-boot.version>

    <maven-compiler-plugin.version>3.3</maven-compiler-plugin.version>
    <maven-surefire-plugin.version>2.18.1</maven-surefire-
plugin.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>io.fabric8</groupId>
        <artifactId>fabric8-project-bom-camel-spring-boot</artifactId>
        <version>${fabric8.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

Notice how the **dependencyManagement** section of the POM references the **fabric8-project-bom-camel-spring-boot** BOM file, which defines the versions for all of the Spring Boot Maven artifacts in FIS. When it comes to patching or upgrading the application, the following version settings are important:

fabric8.version

Defines the version of the **fabric8-project-bom-camel-spring-boot** BOM file. By updating the BOM version to a particular patch version, you are effectively updating all of the FIS Maven dependencies as well.

fabric8.maven.plugin.version

Defines the version of the **fabric8-maven-plugin** plug-in. The **fabric8-maven-plugin** plug-in is tightly integrated with each version of FIS. Hence, whenever you patch or upgrade FIS, it is essential to upgrade the **fabric8-maven-plugin** plug-in to the matching version.

9.2.2. Update Dependencies in a Karaf Application

The following code fragment shows the standard layout of a POM file for a Karaf application in FIS, highlighting some important property settings:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
      <!-- configure the versions you want to use here -->
    <fabric8.version>2.2.170.redhat-000004</fabric8.version>
    <karaf.plugin.version>4.0.8.redhat-000017</karaf.plugin.version>

    <!-- maven plugin versions -->
    ...
    <fabric8.maven.plugin.version>3.1.80.redhat-
000004</fabric8.maven.plugin.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <!-- fabric8 bom must be before fabric8 bom -->
      <dependency>
        <groupId>io.fabric8</groupId>
        <artifactId>fabric8-project-bom-fuse-karaf</artifactId>
        <version>${fabric8.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

Notice how the **dependencyManagement** section of the POM references the **fabric8-project-bom-fuse-karaf** BOM file, which defines the versions for all of the Karaf Maven artifacts in FIS. When it comes to patching or upgrading the application, the following version settings are important:

fabric8.version

Defines the version of the **fabric8-project-bom-fuse-karaf** BOM file. By updating the BOM version to a particular patch version, you are effectively updating all of the FIS Maven dependencies as well.

fabric8.maven.plugin.version

Defines the version of the **fabric8-maven-plugin** plug-in. The **fabric8-maven-plugin** plug-in is tightly integrated with each version of FIS. Hence, whenever you patch or upgrade FIS, it is essential to upgrade the **fabric8-maven-plugin** plug-in to the matching version.

9.3. PATCH THE FIS TEMPLATES

You must update the FIS templates to the latest patch level, to ensure that new template-based projects are built using the correct patched dependencies. Patch the FIS templates as follows:

1. You need administrator privileges to update the FIS templates in OpenShift. Log in to the OpenShift Server as an administrator, as follows:

```
oc login URL -u ADMIN_USER -p ADMIN_PASS
```

Where **URL** is the URL of the OpenShift server and **ADMIN_USER**, **ADMIN_PASS** are the credentials of an administrator account on the OpenShift server.

2. Install the patched FIS templates. Enter the following commands at a command prompt:

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/GA
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-camel-log-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-camel-rest-sql-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf2-cxf-rest-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-config-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-drools-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-infinispan-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-rest-sql-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-teiid-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-xml-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-cxf-jaxrs-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-cxf-jaxws-template.json
```

Note

The **BASEURL** points at the GA branch of the Git repository that stores the quickstart templates and it will always have the latest templates at **HEAD**. So, any time you run the preceding commands, you will get the latest version of the templates.

9.4. PATCH THE FIS IMAGES

The FIS images are updated independently of the main Fuse product. If any patches are required for the FIS images, updated images will be made available on the standard FIS image streams and the updated images can be downloaded from the Red Hat image registry, **registry.access.redhat.com**. Fuse Integration Services provides the following image streams (identified by their OpenShift *image stream name*):

- **fis-java-openshift**
- **fis-karaf-openshift**

These image streams are normally installed on the **openshift** project on the OpenShift server. To check the status of the FIS images on OpenShift, login to OpenShift as an administrator and enter the following command:

```
$ oc get is -n openshift
NAME                                  DOCKER REPO
TAGS                                  UPDATED
fis-java-openshift
registry.access.redhat.com/jboss-fuse-6/fis-java-openshift        2.0-
2,2.0-3,latest + 1 more...     3 days ago
fis-karaf-openshift
registry.access.redhat.com/jboss-fuse-6/fis-karaf-openshift
latest,2.0,2.0-2 + 1 more...       3 days ago
...
```

You can now update each image stream one at a time:

```
oc import-image -n openshift fis-java-openshift
oc import-image -n openshift fis-karaf-openshift
```

Note

You can also configure your Fuse applications so that a rebuild is automatically triggered whenever a new FIS image becomes available. For details, see the section Setting Deployment Triggers in the OpenShift Container Platform 3.3 *Developer Guide*.

APPENDIX A. SPRING BOOT MAVEN PLUG-IN

A.1. SPRING BOOT MAVEN PLUGIN OVERVIEW

This appendix describes the Spring Boot Maven Plugin. It provides the Spring Boot support in Maven and allows you to package the executable jar or war archives and run an application **`in-place`**.

A.2. GOALS

The Spring Boot Plugin includes the following goals:

1. **`spring-boot:run`** runs your Spring Boot application.
2. **`spring-boot:repackage`** repackages your **`.jar`** and **`.war`** files to be executable.
3. **`spring-boot:start`** and **`spring-boot:stop`** both are used to manage the lifecycle of your Spring Boot application.
4. **`spring-boot:build-info`** generates build information that can be used by the Actuator.

A.3. USAGE

You can find general instructions on how to use the Spring Boot Plugin at: `http://docs.spring.io/spring-boot/docs/current/maven-plugin/usage.html`. Following is an example that describes the usage of the `spring-boot-camel' plugin.

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>io.fabric8.quickstarts</groupId>
  <artifactId>spring-boot-camel</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>Fabric8 :: Quickstarts :: Spring-Boot :: Camel</name>
  <description>Spring Boot example running a Camel route</description>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <!-- configure the versions you want to use here -->
    <fabric8.version>2.2.179</fabric8.version>
    <spring-boot.version>1.4.1.RELEASE</spring-boot.version>

    <!-- maven plugin versions -->
    <fabric8.maven.plugin.version>3.2.1</fabric8.maven.plugin.version>
    <maven-compiler-plugin.version>3.3</maven-compiler-plugin.version>
    <maven-surefire-plugin.version>2.18.1</maven-surefire-
plugin.version>
  </properties>

  <dependencyManagement>
```

```
    <dependencies>
      <dependency>
        <groupId>io.fabric8</groupId>
        <artifactId>fabric8-project-bom-camel-spring-boot</artifactId>
        <version>${fabric8.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>

    <!-- Enabling health checks -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-spring-boot-starter</artifactId>
    </dependency>

    <!-- testing -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.jboss.arquillian.junit</groupId>
      <artifactId>arquillian-junit-container</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>io.fabric8</groupId>
      <artifactId>fabric8-arquillian</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

```
  <build>
    <defaultGoal>spring-boot:run</defaultGoal>

    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>${maven-compiler-plugin.version}</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>${maven-surefire-plugin.version}</version>
        <inherited>true</inherited>
        <configuration>
          <excludes>
            <exclude>**/*KT.java</exclude>
          </excludes>
        </configuration>
      </plugin>

      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>${spring-boot.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>

      <plugin>
        <groupId>io.fabric8</groupId>
        <artifactId>fabric8-maven-plugin</artifactId>
        <version>${fabric8.maven.plugin.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>resource</goal>
              <goal>build</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

For more information on Spring Boot Maven Plugin, refer the **http://docs.spring.io/spring-boot/docs/current/maven-plugin** link.

APPENDIX B. KARAF MAVEN PLUG-IN

B.1. USING THE KARAF-MAVEN-PLUGIN

The **karaf-maven-plugin** enables you to create a Karaf *server assembly*, which is a microservices style packaging of a Karaf container. That is, the finished assembly contains all of the essential components of a Karaf installation (for example, including the contents of the **etc/**, **data/**, **lib**, and **system** directories), but stripped down to the bare minimum required to run your application.

B.2. KARAF MAVEN PLUG-IN GOALS

The following Karaf Maven plug-in goals are relevant to building server assemblies in Fuse Integration Services (FIS):

- Section B.2.1, “karaf:assembly Goal”

B.2.1. karaf:assembly Goal

The recommended way to create a Karaf server assembly is to use the **karaf:assembly** goal provided by the **karaf-maven-plugin**. This assembles a server from the Maven dependencies in the project pom.

B.2.1.1. Example of a Maven Assembly

You can create a Karaf server assembly using the **karaf:assembly** goal provided by the **karaf-maven-plugin**. This goal assembles a microservices style server assembly from the Maven dependencies in the project POM. In a FIS project, it is recommended that you bind the **karaf:assembly** goal to the Maven **install** phase. The project uses **bundle** packaging and the project itself gets installed into the Karaf container by listing it inside the **startupBundles** element. The following example displays the typical Maven configuration in a FIS quickstart:

```
<plugin>
  <groupId>org.apache.karaf.tooling</groupId>
  <artifactId>karaf-maven-plugin</artifactId>
  <version>${karaf.plugin.version}</version>
  <extensions>true</extensions>
  <executions>
    <execution>
      <id>karaf-assembly</id>
      <goals>
        <goal>assembly</goal>
      </goals>
      <phase>install</phase>
    </execution>
  </executions>
  <configuration>
    <!-- we are using karaf 2.4.x -->
    <karafVersion>v24</karafVersion>
    <useReferenceUrls>true</useReferenceUrls>
    <archiveTarGz>false</archiveTarGz>
    <includeBuildOutputDirectory>false</includeBuildOutputDirectory>
```

```
    <startupFeatures>
      <feature>karaf-framework</feature>
      <feature>shell</feature>
      <feature>jaas</feature>
      <feature>aries-blueprint</feature>
      <feature>camel-blueprint</feature>
      <feature>fabric8-karaf-blueprint</feature>
      <feature>fabric8-karaf-checks</feature>
    </startupFeatures>
    <startupBundles>

<bundle>mvn:${project.groupId}/${project.artifactId}/${project.version}</
bundle>
    </startupBundles>
  </configuration>
</plugin>
```

B.2.1.2. Parameters

The **karaf:assembly** goal supports the following parameters:

startupFeature

This will result in the feature bundles being listed in **startup.properties** at the appropriate start level and the bundles being copied into the **system/** internal repository. You can use **<feature-name>** or **<feature-name>/<feature-version>** formats.

bootFeature

This will result in the feature name being added to **boot-features** in the features service configuration file and all the bundles in the feature copied into the **system/** internal repository. You can use **<feature-name>** or **<feature-name>/<feature-version>** formats.

installedFeature

This will result in all the bundles in the feature being installed in the **system/** internal repository. Therefore, at run time the feature may be installed without access to external repositories. You can use **<feature-name>** or **<feature-name>/<feature-version>** formats.

libraries

The plugin accepts the **libraries** element, which can have one or more **library** child elements that specify a library URL. For example:

```
<libraries>
    <library>mvn:org.postgresql/postgresql/9.3-1102-
jdbc41;type:=endorsed</library>
</libraries>
```

APPENDIX C. FABRIC8 MAVEN PLUG-IN

C.1. OVERVIEW

With the help of **`fabric8-maven-plugin`**, you can deploy your Java applications to OpenShift. It provides tight integration with Maven and benefits from the build configuration already provided. This plug-in focuses on the following tasks:

- Building Docker-formatted images and,
- Creating OpenShift resource descriptors

It can be configured very flexibly and supports multiple configuration models for creating:

- A *Zero-Config* setup, which allows for a quick ramp-up with some opinionated defaults. Or for more advanced requirements,
- An *XML configuration*, which provides additional configuration options that can be added to the **`pom.xml`** file.

C.1.1. Building Images

The **`fabric8:build`** goal is for creating Docker-formatted images containing an application. It is easy to include build artifacts and their dependencies in these images. This plugin uses the assembly descriptor format from the **`maven-assembly-plugin`** to specify the content which will be added to the image.

Important

Fuse Integration Services supports only the OpenShift **`s2i`** build strategy, *not* the **`docker`** build strategy.

C.1.2. Kubernetes and OpenShift Resources

Kubernetes and OpenShift resource descriptors can be created with **`fabric8:resource`**. These files are packaged within the Maven artifacts and can be deployed to a running orchestration platform with **`fabric8:apply`**.

C.1.3. Configuration

There are three levels of configuration:

- Zero-Config mode helps to make some very useful decisions based on what is present in the **`pom.xml`** file like, what base image to use or which ports to expose. It is used for starting up things and for keeping quickstart applications small and tidy.
- XML plugin configuration mode is similar to what docker-maven-plugin provides. It allows for type safe configuration with IDE support, but only a subset of possible resource descriptor features is provided.
- Kubernetes and OpenShift resource fragments are user provided YAML files that can be enriched by the plugin. This allows expert users to use plain configuration file with all their capabilities, but also to add project specific build information and avoid boilerplate code.

For more information about the Configuration, see **https://maven.fabric8.io/#configuration**.

C.2. INSTALLING THE PLUGIN

The Fabric8 Maven plugin is available under the Maven central repository and can be connected to pre- and post-integration phases as shown below.

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>{version}</version>

  <configuration>
     ....
     <images>
        <!-- A single's image configuration -->
        <image>
          ...
          <build>
           ....
          </build>
        </image>
        ....
     </images>
  </configuration>

  <!-- Connect fabric8:resource and fabric8:build to lifecycle phases -->
  <executions>
    <execution>
       <id>fabric8</id>
       <goals>
         <goal>resource</goal>
         <goal>build</goal>
       </goals>
    </execution>
  </executions>
</plugin>
```

C.3. UNDERSTANDING THE GOALS

The Fabric8 Maven Plugin supports a rich set of goals for providing a smooth Java developer experience. You can categorize these goals as follows:

- Build goals are used to create and manage the Kubernetes and OpenShift build artifacts like Docker-formatted images or S2I builds.
- Development goals are used in deploying resource descriptors to the development cluster. Also, helps you to manage the lifecycle of the development cluster.

The following are the goals supported by the Fabric8 Maven plugin in the Red Hat Fabric Integration Services product:

Table C.1. Build Goals

Goal	Description
fabric8:build	Build images. Note that Fuse Integration Services supports only the OpenShift `s2i` build strategy, not the `docker` build strategy.
fabric8:resource	Create Kubernetes or OpenShift resource descriptors
fabric8:apply	Apply resources to a running cluster
fabric8:resource-apply	Run `fabric8:resource` → `fabric8:apply`

Table C.2. Development Goals

Goal	Description
fabric8:run	Run a complete development workflow cycle `fabric8:resource` → `fabric8:build` → `fabric8:apply` in the foreground.
fabric8:deploy	Deploy resources descriptors to a cluster after creating them and building the app. Same as `fabric8:run` except that it runs in the background.
fabric8:undeploy	Undeploy and remove resources descriptors from a cluster.
fabric8:start	Start the application which has been deployed previously
fabric8:stop	Stop the application which has been deployed previously
fabric8:log	Show the logs of the running application
fabric8:debug	Enable remote debugging

For more information about the Fabric8 Maven plugin goals, see https://maven.fabric8.io/#goals.

C.4. GENERATORS

The Fabric8 Maven plug-in provides *generator* components, which have the capability to build images automatically for specific kinds of application. In the case of Fuse Integration Services, the following generator types are supported:

- Section C.4.3, “Spring Boot”
- Section C.4.4, “Karaf”

Depending on certain characteristics of the application project, the generator framework auto-detects what type of build is required and invokes the appropriate generator component.

Note

The open source community version of the Fabric8 Maven plug-in provides additional generator types, but these are not supported in the Fuse Integration Services product.

C.4.1. Zero-Configuration

Generators do not *require* any configuration. They are enabled by default and run automatically with default settings when the Fabric8 Maven plug-in is invoked. But you can easily customize the configuration of the generators, if you need to.

C.4.2. Modes for Specifying the Base Image

In Fuse Integration Services, the base image for an application build can either be a Java image (for Spring Boot applications) or a Karaf image (for Karaf applications) The Fabric8 Maven plug-in supports the following modes for specifying the base image:

`istag`

(Default) The *image stream* mode works by selecting a tagged image from an OpenShift image stream. In this case, the base image is specified in the following format:

```
<namespace>/<image-stream-name>:<tag>
```

Where **`<namespace>`** is the name of the OpenShift project where the image streams are defined (normally, **`openshift`**), **`<image-stream-name>`** is the name of the image stream, and **`<tag>`** identifies a particular image in the stream (or tracks the *latest* image in the stream).

`docker`

The *docker* mode works by selecting a particular Docker-formatted image directly from an image registry. Because the base image is obtained directly from a remote registry, an image stream is not required. In this case, the base image is specified in the following format:

```
[<registry-location-url>/]<image-namespace>/<image-name>:<tag>
```

Where the image specifier *optionally* begins with the URL location of the remote image registry **<registry-location-url>**, followed by the image namespace **<image-namespace>**, the image name **<image-name>**, and the tag, **<tag>**.

Note

The default behaviour of the open source community version of **fabric8-maven-plugin** is different from the Red Hat productized version (for example, in the community version, the default mode is **docker**).

C.4.2.1. Default Values for istag Mode

When **istag** mode is selected (which is the default), the Fabric8 Maven plug-in uses the following default image specifiers to select the Fuse images (formatted as **<namespace>/<image-stream-name>:<tag>**):

```
openshift/fis-java-openshift:2.0
openshift/fis-karaf-openshift:2.0
```

Note

In the Fuse image streams, the individual images are tagged with build numbers — for example, **2.0-1**, **2.0-2**, and so on. The **2.0** tag is configured to always track the latest image.

C.4.2.2. Default Values for docker Mode

When **docker** mode is selected, and assuming that the OpenShift environment is configured to access **registry.access.redhat.com**, the Fabric8 Maven plug-in uses the following default image specifiers to select the Fuse images (formatted as **<image-namespace>/<image-name>:<tag>**):

```
jboss-fuse-6/fis-java-openshift:2.0
jboss-fuse-6/fis-karaf-openshift:2.0
```

C.4.2.3. Mode Configuration for Spring Boot Applications

To customize the mode configuration and base image location used for building Spring Boot applications, add a **configuration** element to the **fabric8-maven-plugin** configuration in your application's **pom.xml** file, in the following format:

```
<configuration>
  <generator>
    <config>
      <spring-boot>
        <fromMode>{istag|docker}</fromMode>
        <from>{image locations}</from>
```

```
      </spring-boot>
    </config>
  </generator>
</configuration>
```

C.4.2.4. Mode Configuration for Karaf Applications

To customize the mode configuration and base image location used for building Karaf applications, add a **configuration** element to the **fabric8-maven-plugin** configuration in your application's **pom.xml** file, in the following format:

```
<configuration>
  <generator>
    <config>
      <karaf>
        <fromMode>{istag|docker}</fromMode>
        <from>{image locations}</from>
      </karaf>
    </config>
  </generator>
</configuration>
```

C.4.2.5. Specifying the Mode on the Command Line

As an alternative to customizing the mode configuration directly in the **pom.xml** file, you can pass the mode settings directly to the **mvn** command, by adding the following property settings to the command line invocation:

```
//build from Docker-formatted image directly, registry location, image
name or tag are subject to change if desirable
-Dfabric8.generator.fromMode=docker
-Dfabric8.generator.from=<custom-registry-location-url>/<image-
namespace>/<image-name>:<tag>

//to use ImageStream from different namespace
-Dfabric8.generator.fromMode=istag //istag is default
-Dfabric8.generator.from=<namespace>/<image-stream-name>:<tag>
```

C.4.3. Spring Boot

The Spring Boot generator gets activated when it finds a **spring-boot-maven-plugin** plug-in in the **pom.xml** file. The generated container port is read from the **server.port** property **application.properties**, defaulting to **8080** if it is not found.

In addition to the common generator options, this generator can be configured with the following options:

Table C.3. Spring-Boot configuration options

Element	Description	Default
assemblyRef	If a reference to an assembly is given, then this is used without trying to detect the artifacts to include.	
targetDir	Directory within the generated image where the detected artefacts are put. Change this only if the base image is changed too.	**`/deployments`**
jolokiaPort	Port of the Jolokia agent exposed by the base image. Set this to 0 if you don't want to expose the Jolokia port.	8778
mainClass	Main class to call. If not specified, the generator searches for the main class as follows. First, a check is performed to detect a fat-jar. Next, the **`target/classes`** directory is scanned to look for a single class with a **`main`** method. If none is found or more than one is found, the generator does nothing.	
webPort	Port to expose as service, which is supposed to be the port of a web application. Set this to 0 if you don't want to expose a port.	8080
color	If set, force the use of color in the Spring Boot console output.	

The generator adds Kubernetes liveness and readiness probes pointing to either the management or server port as read from the **`application.properties`**. If the **`server.ssl.key-store`** property is set in **`application.properties`** then the probes are automatically set to use **`https`**.

C.4.4. Karaf

The Karaf generator gets activated when it finds a **`karaf-maven-plugin`** plug-in in the **`pom.xml`** file.

In addition to the common generator options, this generator can be configured with the following options:

Table C.4. Karaf configuration options

Element	Description	Default
baseDir	Directory within the generated image where the detected artifacts are put. Change this only if the base image is changed too.	**`/deployments`**

Element	Description	Default
jolokiaPort	Port of the Jolokia agent exposed by the base image. Set this to 0 if you don't want to expose the Jolokia port.	8778
mainClass	Main class to call. If not specified, the generator searches for the main class as follows. First, a check is performed to detect a fat-jar. Next, the **`target/classes`** directory is scanned to look for a single class with a **`main`** method. If none is found or more than one is found, the generator does nothing.	
user	User and/or group under which the files should be added. The user must already exist in the base image. It has the general format **`<user>[:<group>[:<run-user>]]`**`. The user and group can be given either as numeric user- and group-id or as names. The group id is optional.	**`jboss:jboss:jboss`**
webPort	Port to expose as service, which is supposed to be the port of a web application. Set this to 0 if you don't want to expose a port.	8080

APPENDIX D. FABRIC8 CAMEL MAVEN PLUG-IN

D.1. GOALS

For validating Camel endpoints in the source code:

- **fabric8-camel:validate** validates the Maven project source code to identify invalid camel endpoint uris

D.2. ADDING THE PLUGIN TO YOUR PROJECT

To enable the Plugin, add the following to the **pom.xml** file:

```
<plugin>
  <groupId>io.fabric8.forge</groupId>
  <artifactId>fabric8-camel-maven-plugin</artifactId>
  <version>2.3.52</version>
</plugin>
```

Note: Check the current version number of the fabric8-forge release. You can find the latest release at the following location: **https://github.com/fabric8io/fabric8-forge/releases**.

However, you can run the validate goal from the command line or from your Java editor such as IDEA or Eclipse.

```
mvn fabric8-camel:validate
```

You can also enable the Plugin to run automatically as a part of the build to catch the errors.

```
<plugin>
  <groupId>io.fabric8.forge</groupId>
  <artifactId>fabric8-camel-maven-plugin</artifactId>
  <version>2.3.80</version>
  <executions>
    <execution>
      <phase>process-classes</phase>
    <goals>
      <goal>validate</goal>
    </goals>
    </execution>
  </executions>
</plugin>
```

The phase determines when the Plugin runs. In the above example, the phase is **process-classes** which runs after the compilation of the main source code.

You can also configure the maven plugin to validate the test source code. Change the phase as per the **process-test-classes** as shown below:

```
<plugin>
  <groupId>io.fabric8.forge</groupId>
  <artifactId>fabric8-camel-maven-plugin</artifactId>
```

```
  <version>2.3.80</version>
  <executions>
    <execution>
      <configuration>
        <includeTest>true</includeTest>
      </configuration>
      <phase>process-test-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

D.3. RUNNING THE GOAL ON ANY MAVEN PROJECT

You can also run the validate goal on any Maven project, without adding the Plugin to the **pom.xml** file. You need to specify the Plugin, using its fully qualified name. For example, to run the goal on the camel-example-cdi plugin from Apache Camel, execute the following:

```
    $cd camel-example-cdi
    $mvn io.fabric8.forge:fabric8-camel-maven-plugin:2.3.52:validate
```

which then runs and displays the following output:

```
[INFO] ------------------------------------------------------------
------------
[INFO] Building Camel :: Example :: CDI 2.16.2
[INFO] ------------------------------------------------------------
------------
[INFO]
[INFO] --- fabric8-camel-maven-plugin:2.3.52:validate (default-cli) @
camel-example-cdi ---
[INFO] Endpoint validation success: (4 = passed, 0 = invalid, 0 =
incapable, 0 = unknown components)
[INFO] Simple validation success: (0 = passed, 0 = invalid)
[INFO] ------------------------------------------------------------
------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
------------
```

After passing the validation successfully, you can validate the four endpoints. Let us assume that you made a typo in one of the Camel endpoint uris in the source code, such as:

```
  @Uri("timer:foo?period=5000")
```

You can make changes to include a typo error in the **period** option, such as:

```
  @Uri("timer:foo?perid=5000")
```

And when running the validate goal again, reports the following:

```
[INFO] ------------------------------------------------------------
```

```
------------
[INFO] Building Camel :: Example :: CDI 2.16.2
[INFO] -------------------------------------------------------------
------------
[INFO]
[INFO] --- fabric8-camel-maven-plugin:2.3.52:validate (default-cli) @
camel-example-cdi ---
[WARNING] Endpoint validation error at:
org.apache.camel.example.cdi.MyRoutes(MyRoutes.java:32)

 timer:foo?perid=5000

                    perid    Unknown option. Did you mean: [period]


[WARNING] Endpoint validation error: (3 = passed, 1 = invalid, 0 =
incapable, 0 = unknown components)
[INFO] Simple validation success: (0 = passed, 0 = invalid)
[INFO] -------------------------------------------------------------
------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------------------------
------------
```

D.4. OPTIONS

The maven plugin supports the following options which you can configure from the command line (use **`-D`** syntax), or defined in the **`pom.xml`** file in the **`<configuration>`** tag.

D.4.1. Table

Parameter	Default Value	Description
downloadVersion	true	Whether to allow downloading Camel catalog version from the internet. This is needed, if the project uses a different Camel version than this plugin is using by default.
failOnError	false	Whether to fail if invalid Camel endpoints was found. By default the plugin logs the errors at **`WARN`** level
logUnparseable	false	Whether to log endpoint URIs which was un-parsable and therefore not possible to validate

Parameter	Default Value	Description
includeJava	true	Whether to include Java files to be validated for invalid Camel endpoints
includeXML	true	Whether to include XML files to be validated for invalid Camel endpoints
includeTest	false	Whether to include test source code
includes	-	To filter the names of java and xml files to only include files matching any of the given list of patterns (wildcard and regular expression). Multiple values can be separated by comma.
excludes	-	To filter the names of java and xml files to exclude files matching any of the given list of patterns (wildcard and regular expression). Multiple values can be separated by comma.
ignoreUnknownComponent	true	Whether to ignore unknown components
ignoreIncapable	true	Whether to ignore incapable of parsing the endpoint uri
ignoreLenientProperties	true	Whether to ignore components that uses lenient properties. When this is true, then the uri validation is stricter but would fail on properties that are not part of the component but in the uri because of using lenient properties. For example using the HTTP components to provide query parameters in the endpoint uri.
showAll	false	Whether to show all endpoints and simple expressions (both invalid and valid).

D.5. VALIDATING INCLUDE TEST

If you have a Maven project, then you can run the plugin to validate the endpoints in the unit test source code as well. You can pass in the options using **-D** style as shown:

```
    $cd myproject
    $mvn io.fabric8.forge:fabric8-camel-maven-plugin:2.3.52:validate -
DincludeTest=true
```

APPENDIX E. JVM ENVIRONMENT VARIABLES

E.1. S2I JAVA BUILDER IMAGE WITH OPENJDK 8

In this S2I builder image for Java builds, you can run results directly without using any other application server. It is suitable for microservices with a flat classpath (including **fat jars**).

You can configure java options when using the FIS images. All the options for the FIS images are set by using environment variables as given below. For the JVM options, you can use the environment variable **JAVA_OPTIONS**. Also, provide **JAVA_ARGS** for the arguments which are given through to the application.

E.2. S2I KARAF BUILDER IMAGE WITH OPENJDK 8

This image can be used with OpenShift's Source To Image in order to build Karaf4 custom assembly based maven projects.

Following is the command to use S2I:

```
s2i build <git repo url> registry.access.redhat.com/jboss-fuse-6/fis-
karaf-openshift:2.0 <target image name> docker run <target image name>
```

E.2.1. Configuring the Karaf4 Assembly

The location of the Karaf4 assembly built by the maven project can be provided in multiple ways.

- Default assembly file ***.tar.gz** in output directory
- By using the **-e flag** in sti or oc command
- By setting **FUSE_ASSEMBLY** property in **.sti/environment** under the project source

E.2.2. Customizing the Build

It is possible to customize the maven build. The **MAVEN_ARGS** environment variable can be set to change the behaviour.

By default, the **MAVEN_ARGS** is set as follows:

```
Karaf4: install karaf:assembly karaf:archive -DskipTests -e
```

E.3. ENVIRONMENT VARIABLES

Following are the environment variables that are used to influence the behaviour of S2I Java and Karaf builder images:

E.3.1. Build Time

During the build time, you can use the following environment variables:

- **MAVEN_ARGS**: Arguments to use when calling maven, replacing the default package.

- **MAVEN_ARGS_APPEND**: Additional Maven arguments, useful for adding temporary arguments like **-X** or **-am -pl**.
- **ARTIFACT_DIR**: Path to **target/** where the jar files are created for multi-module builds. These are added to **${MAVEN_ARGS}**.
- **ARTIFACT_COPY_ARGS**: Arguments to use when copying artifacts from the output directory to the application directory. Useful to specify which artifacts will be part of the image.
- **MAVEN_CLEAR_REPO**: If set, remove the Maven repository after you build the artifact. This is useful for keeping the application image small, however, It prevents the incremental builds. The default value is false.

E.3.2. Run Time

You can use the following environment variables to influence the run script:

- **JAVA_APP_DIR**: the directory where the application resides. All paths in your application are relative to the directory.
- **JAVA_LIB_DIR**: this directory contains the Java jar files as well an optional classpath file, which holds the classpath. Either as a single line classpath (colon separated) or with jar files listed line-by-line. However, If not set, then **JAVA_LIB_DIR** is the same as **JAVA_APP_DIR** directory.
- **JAVA_OPTIONS**: options to add when calling java.
- **JAVA_MAX_MEM_RATIO**: It is used when no **-Xmx** option is given in JAVA_OPTIONS. This is used to calculate a default maximal heap Memory based on a containers restriction. If used in a Docker container without any memory constraints for the container, then this option has no effect.
- **JAVA_MAX_CORE**: It restricts manually the number of cores available, which is used for calculating certain defaults like the number of garbage collector threads. If set to 0, you cannot perform the base JVM tuning based on the number of cores.
- **JAVA_DIAGNOSTICS**: Set this to fetch some diagnostics information, to standard out when things are happening.
- **JAVA_MAIN_CLASS**: A main class to use as an argument for java. When you give this environment variable, all jar files in **$JAVA_APP_DIR** directory are added to the classpath and in the **$JAVA_LIB_DIR** directory.
- **JAVA_APP_JAR**: A jar file with an appropriate manifest, so that you can start with **java -jar**. However, if it is not provided, then **$JAVA_MAIN_CLASS** is set. In all cases, this jar file is added to the classpath.
- **JAVA_APP_NAME**: Name to use for the process.
- **JAVA_CLASSPATH**: the classpath to use. If not given, the startup script checks for a file **${JAVA_APP_DIR}/classpath** and use its content as classpath. If this file doesn't exists, then all jars in the application directory are added under **(classes:${JAVA_APP_DIR}/*)**.
- **JAVA_DEBUG**: If set, remote debugging will be switched on.
- **JAVA_DEBUG_PORT**: Port used for remote debugging. The default value is 5005.

E.3.3. Jolokia Configuration

You can use the following environment variables in Jolokia:

- **AB_JOLOKIA_OFF**: If set, disables the activation of Jolokia (echos an empty value). By default, Jolokia is enabled.
- **AB_JOLOKIA_CONFIG**: If set, uses the file (including path) as Jolokia JVM agent properties. However, If not set, the **/opt/jolokia/etc/jolokia.properties** will be created using the settings.
- **AB_JOLOKIA_HOST**: Host address to bind (Default value is 0.0.0.0)
- **AB_JOLOKIA_PORT**: Port to use (Default value is 8778)
- **AB_JOLOKIA_USER**: User for basic authentication. By default, it is **jolokia**
- **AB_JOLOKIA_PASSWORD**: Password for basic authentication. By default, authentication is switched off
- **AB_JOLOKIA_PASSWORD_RANDOM**: Generates a value and is written in **/opt/jolokia/etc/jolokia.pw** file
- **AB_JOLOKIA_HTTPS**: Switch on secure communication with **HTTPS**. By default, self-signed server certificates are generated, if no serverCert configuration is given in **AB_JOLOKIA_OPTS**
- **AB_JOLOKIA_ID**: Agent ID to use
- **AB_JOLOKIA_DISCOVERY_ENABLED**: Enables the Jolokia discovery. The default value is false.
- **AB_JOLOKIA_OPTS**: Additional options to be appended to the agent configuration. Options are given in the format **key=value**

Here is an option for integration with various environments:

- **AB_JOLOKIA_AUTH_OPENSHIFT**: Switch on client authentication for OpenShift TSL communication. Ensure that the value of this parameter must be present in a client certificate. If you enable this parameter, it will automatically switch Jolokia into **HTTPS** communication mode. The default CA cert is set to **/var/run/secrets/kubernetes.io/serviceaccount/ca.crt**

Application arguments can be provided by setting the variable **JAVA_ARGS** to the corresponding value.