Red Hat Fuse 7.4

Fuse on OpenShift Guide

Installing and developing with Red Hat Fuse on OpenShift

Last Updated: 2020-05-29

Red Hat Fuse 7.4 Fuse on OpenShift Guide

Installing and developing with Red Hat Fuse on OpenShift

Legal Notice

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Guide to using Fuse on OpenShift

Table of Contents

PREFACE 6

CHAPTER 1. BEFORE YOU BEGIN 7
1.1. COMPARISON: FUSE STANDALONE AND FUSE ON OPENSHIFT 7

CHAPTER 2. GETTING STARTED FOR ADMINISTRATORS 9
2.1. CONFIGURING RED HAT CONTAINER REGISTRY AUTHENTICATION 9
2.2. INSTALLING FUSE IMAGESTREAMS AND TEMPLATES ON THE OPENSHIFT 4.X SERVER 10
2.3. INSTALLING APICURITO OPERATOR ON THE OPENSHIFT 4.X SERVER 12
2.4. INSTALLING FUSE IMAGESTREAMS AND TEMPLATES ON THE OPENSHIFT 3.X SERVER 13

CHAPTER 3. INSTALLING FUSE ON OPENSHIFT AS A NON-ADMIN USER 16
3.1. INSTALLING FUSE ON OPENSHIFT IMAGES AND TEMPLATES AS A NON-ADMIN USER 16

CHAPTER 4. GETTING STARTED FOR DEVELOPERS 18
4.1. PREPARING DEVELOPMENT ENVIRONMENT 18
4.1.1. Installing Container Development Kit (CDK) on Your Local Machine 18
4.1.2. Getting Remote Access to an Existing OpenShift Server 19
4.1.3. Installing Client-Side Tools 19
4.1.4. Configuring Maven Repositories 20
4.2. CREATING AND DEPLOYING APPLICATIONS ON FUSE ON OPENSHIFT 20
4.2.1. Creating and Deploying a Project Using the S2I Binary Workflow 20
4.2.2. Undeploying and Redeploying the Project 24
4.2.3. Set up Fuse Console on OpenShift 25
4.2.3.1. Monitoring a single Fuse pod from the Fuse Console 26
4.2.3.2. Deploying the Fuse Console from the OpenShift Console 27
4.2.3.3. Deploying the Fuse Console from the command line 28
4.2.3.4. Ensuring that data displays correctly in the Fuse Console 29
4.2.4. Creating and Deploying a Project Using the S2I Source Workflow 29

CHAPTER 5. DEVELOPING AN APPLICATION FOR THE SPRING BOOT IMAGE 33
5.1. CREATING A SPRING BOOT PROJECT USING MAVEN ARCHETYPE 33
5.2. STRUCTURE OF THE CAMEL SPRING BOOT APPLICATION 33
5.3. SPRING BOOT ARCHETYPE CATALOG 35
5.4. BOM FILE FOR SPRING BOOT 37
5.4.1. Incorporate the BOM file 37
5.5. SPRING BOOT MAVEN PLUGIN 39

CHAPTER 6. APACHE CAMEL IN SPRING BOOT 40
6.1. INTRODUCTION TO CAMEL SPRING BOOT 40
6.2. INTRODUCTION TO CAMEL SPRING BOOT STARTER 40
6.3. AUTO-CONFIGURED CAMEL CONTEXT 41
6.4. AUTO-DETECTING CAMEL ROUTES 42
6.5. CAMEL PROPERTIES 42
6.6. CUSTOM CAMEL CONTEXT CONFIGURATION 43
6.7. DISABLING JMX 43
6.8. AUTO-CONFIGURED CONSUMER AND PRODUCER TEMPLATES 43
6.9. AUTO-CONFIGURED TYPECONVERTER 44
6.10. SPRING TYPE CONVERSION API BRIDGE 44
6.11. DISABLING TYPE CONVERSIONS FEATURES 45
6.12. ADDING XML ROUTES 45
6.13. ADDING XML REST-DSL 46
6.14. TESTING WITH CAMEL SPRING BOOT 46

6.15. SEE ALSO 47

CHAPTER 7. RUNNING A CAMEL SERVICE ON SPRING BOOT WITH XA TRANSACTIONS 48
7.1. STATEFULSET RESOURCES 48
7.2. SPRING BOOT NARAYANA RECOVERY CONTROLLER 48
7.3. CONFIGURING SPRING BOOT NARAYANA RECOVERY CONTROLLER 48
7.4. RUNNING CAMEL SPRING BOOT XA QUICKSTART ON OPENSHIFT 49
7.5. TESTING SUCCESSFUL XA TRANSACTIONS 51
7.6. TESTING FAILED XA TRANSACTIONS 51

CHAPTER 8. INTEGRATING A CAMEL APPLICATION WITH THE A-MQ BROKER 52
8.1. BUILDING AND DEPLOYING A SPRING BOOT CAMEL A-MQ QUICKSTART 52

CHAPTER 9. INTEGRATING SPRING BOOT WITH KUBERNETES 55
9.1. SPRING BOOT EXTERNALIZED CONFIGURATION 55
9.1.1. Kubernetes ConfigMap 55
9.1.2. Kubernetes Secrets 55
9.1.3. Spring Cloud Kubernetes Plug-In 55
9.1.4. Enabling Spring Boot with Kubernetes Integration 55
9.2. RUNNING TUTORIAL FOR CONFIGMAP PROPERTY SOURCE 56
9.2.1. Running Spring Boot Camel Config Quickstart 56
9.2.2. Configuration Properties Bean 58
9.2.3. Setting up Secret 60
9.2.4. Setting up ConfigMap 62
9.3. USING CONFIGMAP PROPERTYSOURCE 64
9.3.1. Applying Individual Properties 64
9.3.2. Applying Property Named application.yaml 64
9.3.3. Applying Property Named application.properties 64
9.3.4. Deploying a ConfigMap 65
9.4. USING SECRETS PROPERTYSOURCE 65
9.4.1. Example of Setting Secrets 65
9.4.2. Consuming the Secrets 66
9.4.3. Configuration Properties for Secrets PropertySource 67
9.5. USING PROPERTYSOURCE RELOAD 67
9.5.1. Enabling PropertySource Reload 67
9.5.2. Levels of PropertySource Reload 67
9.5.3. Example of PropertySource Reload 68
9.5.4. PropertySource Reload Operating Modes 69
9.5.5. PropertySource Reload Configuration Properties 69

CHAPTER 10. DEVELOPING AN APPLICATION FOR THE KARAF IMAGE 70
10.1. CREATING A KARAF PROJECT USING MAVEN ARCHETYPE 70
10.2. STRUCTURE OF THE CAMEL KARAF APPLICATION 70
10.3. KARAF ARCHETYPE CATALOG 71
10.4. USING FABRIC8 KARAF FEATURES 72
10.4.1. Adding Fabric8 Karaf Features 72
10.4.2. Adding Fabric8 Karaf Core Bundle Functionality 73
10.4.3. Setting the Property Placeholder Service Options 73
10.4.4. Adding a Custom Property Placeholder Resolver 74
10.4.5. List of Resolution Strategies 75
10.4.6. List of Property Placeholder Service Options 76
10.5. ADDING FABRIC8 KARAF CONFIG ADMIN SUPPORT 77
10.5.1. Adding Fabric8 Karaf Config Admin Support 77
10.5.2. Adding ConfigMap Injection 77

10.5.3. Configuration plugin 78
10.5.4. Config Property Placeholders 78
10.5.5. Fabric8 Karaf Config Admin options 78
10.6. ADDING FABRIC8 KARAF BLUEPRINT SUPPORT 79
10.7. ENABLING FABRIC8 KARAF HEALTH CHECKS 80
10.8. ADDING CUSTOM HEALTH CHECKS 81

CHAPTER 11. DEVELOPING AN APPLICATION FOR THE JBOSS EAP IMAGE 83
11.1. CREATING A JBOSS EAP PROJECT USING THE S2I SOURCE WORKFLOW 83
11.2. STRUCTURE OF THE JBOSS EAP APPLICATION 86
11.3. JBOSS EAP QUICKSTART TEMPLATES 86

CHAPTER 12. USING PERSISTENT STORAGE IN FUSE ON OPENSHIFT 88
12.1. ABOUT VOLUMES AND VOLUME TYPES 88
12.2. ABOUT PERSISTENTVOLUMES 88
12.3. CONFIGURING PERSISTENT VOLUME 88
12.4. CREATING PERSISTENTVOLUMECLAIMS 89
12.5. USING PERSISTENT VOLUMES IN PODS 89

CHAPTER 13. PATCHING FUSE ON OPENSHIFT 91
13.1. IMPORTANT NOTE ON BOMS AND MAVEN DEPENDENCIES 91
13.2. PATCHING THE FUSE ON OPENSHIFT IMAGES 91
13.3. PATCHING THE FUSE ON OPENSHIFT TEMPLATES 92
13.4. PATCH APPLICATION DEPENDENCIES USING BOM 93
13.4.1. Updating Dependencies in a Spring Boot Application 93
13.4.2. Updating Dependencies in a Karaf Application 95
13.4.3. Updating Dependencies in a JBoss EAP Application 96
13.5. AVAILABLE BOM VERSIONS 96

APPENDIX A. SPRING BOOT MAVEN PLUG-IN 97
A.1. SPRING BOOT MAVEN PLUGIN OVERVIEW 97
A.2. GOALS 97
A.3. USAGE 97
A.3.1. Spring Boot Maven Plugin for Spring Boot 2 97
A.3.2. Spring Boot Maven Plugin for Spring Boot 1 99

APPENDIX B. USING KARAF MAVEN PLUGIN 101
B.1. MAVEN DEPENDENCIES 101
B.1.1. Maven dependency scopes 101
B.2. KARAF MAVEN PLUGIN CONFIGURATION 101
B.3. CUSTOMIZED KARAF ASSEMBLY 102
B.3.1. karaf:assembly goal 102

APPENDIX C. FABRIC8 MAVEN PLUG-IN 104
C.1. OVERVIEW 104
C.1.1. Building Images 104
C.1.2. Kubernetes and OpenShift Resources 104
C.1.3. Configuration 104
C.2. INSTALLING THE PLUGIN 105
C.3. UNDERSTANDING THE GOALS 105
C.3.1. Understanding Build and Development Goals: 105
C.3.2. Setting Environmental Variable 106
C.3.3. Resource Validation Configuration 107
C.4. GENERATORS 107
C.4.1. Zero-Configuration 107

C.4.2. Modes for Specifying the Base Image 108
C.4.2.1. Default Values for istag Mode 108
C.4.2.2. Default Values for docker Mode 109
C.4.2.3. Mode Configuration for Spring Boot Applications 109
C.4.2.4. Mode Configuration for Karaf Applications 109
C.4.2.5. Specifying the Mode on the Command Line 109
C.4.3. Spring Boot 110
C.4.4. Karaf 110

APPENDIX D. FABRIC8 CAMEL MAVEN PLUG-IN **112**
D.1. GOALS 112
D.2. ADDING THE PLUGIN TO YOUR PROJECT 112
D.3. RUNNING THE GOAL ON ANY MAVEN PROJECT 113
D.4. OPTIONS 114
D.4.1. Table 114
D.5. VALIDATING INCLUDE TEST 115

APPENDIX E. JVM ENVIRONMENT VARIABLES **116**
E.1. S2I JAVA BUILDER IMAGE WITH OPENJDK 8 116
E.2. S2I KARAF BUILDER IMAGE WITH OPENJDK 8 116
E.2.1. Configuring the Karaf4 Assembly 116
E.2.2. Customizing the Build 116
E.3. ENVIRONMENT VARIABLES 116
E.3.1. Build Time 116
E.3.2. Run Time 117
E.3.3. Jolokia Configuration 118

APPENDIX F. TUNING JVMS TO RUN IN LINUX CONTAINERS **119**
F.1. OVERVIEW 119
F.2. TUNING THE JVM 119
F.3. DEFAULT BEHAVIOUR OF FUSE ON OPENSHIFT IMAGES 119
F.4. CUSTOM TUNING OF FUSE ON OPENSHIFT IMAGES 119
F.5. TUNING THIRD-PARTY LIBRARIES 120

PREFACE

Red Hat Fuse on OpenShift enables you to deploy Fuse applications on OpenShift Container Platform.

CHAPTER 1. BEFORE YOU BEGIN

Release Notes

See the Release Notes for important information about this release.

Version Compatibility and Support

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

Support for Windows O/S

The developer tooling (**oc** client and Container Development Kit) for Fuse on OpenShift 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.

1.1. COMPARISON: FUSE STANDALONE AND FUSE ON OPENSHIFT

There are several major functionality differences:

- An application deployment with Fuse on OpenShift consists of an application and all required runtime components packaged inside a Docker image. Applications are not deployed to a runtime as with Fuse Standalone, the application image itself is a complete runtime environment deployed and managed through OpenShift.
- Patching in an OpenShift environment is different from Fuse Standalone, as each application image is a complete runtime environment. To apply a patch, the application image is rebuilt and redeployed within OpenShift. Core OpenShift management capabilities allow for rolling upgrades and side-by-side deployment to maintain availability of your application during upgrade.
- Provisioning and clustering capabilities provided by Fabric in Fuse have been replaced with equivalent functionality in Kubernetes and OpenShift. There is no need to create or configure individual child containers as OpenShift automatically does this for you as part of deploying and scaling your application.
- Fabric endpoints are not used within an OpenShift environment. Kubernetes services must be used instead.
- Messaging services are created and managed using the A-MQ for OpenShift image and *not* included directly within a Karaf container. Fuse on OpenShift provides an enhanced version of the camel-amq component to allow for seamless connectivity to messaging services in OpenShift through Kubernetes.
- Live updates to running Karaf instances using the Karaf shell is strongly discouraged as updates will not be preserved if an application container is restarted or scaled up. This is a fundamental tenet of immutable architecture and essential to achieving scalability and flexibility within OpenShift.
- Maven dependencies directly linked to Red Hat Fuse components are supported by Red Hat. Third-party Maven dependencies introduced by users are not supported.
- The SSH Agent is not included in the Apache Karaf micro-container, so you cannot connect to it using the bin/client console client.

- Protocol compatibility and Camel components within a Fuse on OpenShift application: non-HTTP based communications must use TLS and SNI to be routable from outside OpenShift into a Fuse service (Camel consumer endpoint).

CHAPTER 2. GETTING STARTED FOR ADMINISTRATORS

If you are an OpenShift administrator, you can prepare an OpenShift cluster for Fuse on OpenShift deployments by:

1. Configuring authentication to the Red Hat Container Registry.

2. Installing the Fuse on OpenShift images and templates.

2.1. CONFIGURING RED HAT CONTAINER REGISTRY AUTHENTICATION

You must configure authentication to Red Hat container registry before you can import and use the Red Hat Fuse on OpenShift image streams and templates.

Procedure

1. Log in to the OpenShift Server as an administrator:

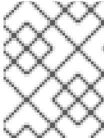
   ```
   oc login -u system:admin
   ```

2. Log in to the OpenShift project where you want to install the image streams. We recommend that you use the **openshift** project for the Fuse on OpenShift image streams.

   ```
   oc project openshift
   ```

3. Create a docker-registry secret using either your Red Hat Customer Portal account or your Red Hat Developer Program account credentials. Replace **<pull_secret_name>** with the name of the secret that you wish to create.

   ```
   oc create secret docker-registry <pull_secret_name> \
     --docker-server=registry.redhat.io \
     --docker-username=CUSTOMER_PORTAL_USERNAME \
     --docker-password=CUSTOMER_PORTAL_PASSWORD \
     --docker-email=EMAIL_ADDRESS
   ```

   NOTE

   You need to create a docker-registry secret in every new namespace where the image streams reside and in every namespace that uses registry.redhat.io.

4. To use the secret for pulling images for pods, add the secret to your service account. The name of the service account must match the name of the service account pod uses. Following example uses **default** which is the default service account.

   ```
   oc secrets link default <pull_secret_name> --for=pull
   ```

5. To use the secret for pushing and pulling build images, the secret must be mountable inside of a pod. To mount the secret, use following command:

   ```
   oc secrets link builder <pull_secret_name>
   ```

If you do not want to use your Red Hat account username and password to create the secret, you should create an authentication token by using a registry service account.

For more information see:

- Red Hat Container Registry Authentication
- Accessing and Configuring the Red Hat Registry

2.2. INSTALLING FUSE IMAGESTREAMS AND TEMPLATES ON THE OPENSHIFT 4.X SERVER

Openshift Container Platform 4.1 uses the Samples Operator, which operates in the OpenShift namespace, installs and updates the Red Hat Enterprise Linux (RHEL)-based OpenShift Container Platform imagestreams and templates. To install the Fuse on OpenShift imagestreams and templates:

- Reconfigure the Samples Operator
- Add Fuse imagestreams and templates to **Skipped Imagestreams and Skipped Templates** fields.
    - Skipped Imagestreams: Imagestreams that are in the Samples Operator's inventory, but that the cluster administrator wants the Operator to ignore or not manage.
    - Skipped Templates: Templates that are in the Samples Operator's inventory, but that the cluster administrator wants the Operator to ignore or not manage.

Prerequisites

- You have access to OpenShift Server.
- You have configured authentication to the Red Hat Container Registry.
- Optionally, if you want the Fuse templates to be visible in the OpenShift dashboard after you install them, you must first install the service catalog and the template service broker as described in the OpenShift documentation (https://docs.openshift.com/container-platform/4.1/applications/service_brokers/installing-service-catalog.html).

Procedure

1. Start the OpenShift 4 Server.
2. Log in to the OpenShift Server as an administrator.

```
oc login -u system:admin
```

3. Verify that you are using the project for which you created a docker-registry secret.

```
oc project openshift
```

4. View the current configuration of Samples operator.

```
oc get configs.samples.operator.openshift.io -n openshift-cluster-samples-operator -o yaml
```

5. Configure Samples operator to ignore the fuse templates and image streams that are added.

```
oc edit configs.samples.operator.openshift.io -n openshift-cluster-samples-operator
```

6. Add the Fuse imagestreams and templates to Skipped Imagestreams and Skipped Templates section respectively.

```
[...]
spec:
  architectures:
  - x86_64
  managementState: Managed
  skippedImagestreams:
  - fis-java-openshift
  - fis-karaf-openshift
  - fuse7-console
  - fuse7-eap-openshift
  - fuse7-java-openshift
  - fuse7-karaf-openshift
  - jboss-fuse70-console
  - jboss-fuse70-eap-openshift
  - jboss-fuse70-java-openshift
  - jboss-fuse70-karaf-openshift
  - fuse-apicurito-generator
  - apicurito-ui
  skippedTemplates:
  - s2i-fuse74-eap-camel-amq
  - s2i-fuse74-eap-camel-cdi
  - s2i-fuse74-eap-camel-cxf-jaxrs
  - s2i-fuse74-eap-camel-cxf-jaxws
  - s2i-fuse74-eap-camel-jpa
  - s2i-fuse74-karaf-camel-amq
  - s2i-fuse74-karaf-camel-log
  - s2i-fuse74-karaf-camel-rest-sql
  - s2i-fuse74-karaf-cxf-rest
  - s2i-fuse74-spring-boot-camel
  - s2i-fuse74-spring-boot-camel-amq
  - s2i-fuse74-spring-boot-camel-config
  - s2i-fuse74-spring-boot-camel-drools
  - s2i-fuse74-spring-boot-camel-infinispan
  - s2i-fuse74-spring-boot-camel-rest-sql
  - s2i-fuse74-spring-boot-camel-xa
  - s2i-fuse74-spring-boot-camel-xml
  - s2i-fuse74-spring-boot-cxf-jaxrs
```

7. Install Fuse on OpenShift image streams.

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003

oc create -n openshift -f ${BASEURL}/fis-image-streams.json
```

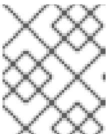
8. Install the quickstart templates:

```
for template in eap-camel-amq-template.json \
```

```
 eap-camel-cdi-template.json \
 eap-camel-cxf-jaxrs-template.json \
 eap-camel-cxf-jaxws-template.json \
 eap-camel-jpa-template.json \
 karaf-camel-amq-template.json \
 karaf-camel-log-template.json \
 karaf-camel-rest-sql-template.json \
 karaf-cxf-rest-template.json \
 spring-boot-camel-amq-template.json \
 spring-boot-camel-config-template.json \
 spring-boot-camel-drools-template.json \
 spring-boot-camel-infinispan-template.json \
 spring-boot-camel-rest-sql-template.json \
 spring-boot-camel-template.json \
 spring-boot-camel-xa-template.json \
 spring-boot-camel-xml-template.json \
 spring-boot-cxf-jaxrs-template.json \
 spring-boot-cxf-jaxws-template.json ;
 do
 oc create -n openshift -f \
 https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003/quickstarts/${template}
 done
```

9. Install the templates for the Fuse Console.

```
oc create -n openshift -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003/fis-console-cluster-template.json
oc create -n openshift -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003/fis-console-namespace-template.json
```



NOTE

For information on deploying the Fuse Console, see Set up Fuse Console on OpenShift.

10. *(Optional)* View the installed Fuse on OpenShift images and templates:

```
oc get template -n openshift
```

2.3. INSTALLING APICURITO OPERATOR ON THE OPENSHIFT 4.X SERVER

Red Hat Fuse on OpenShift provides Apicurito, a web-based API designer, that you can use to design REST APIs. The new Apicurito Operator simplifies the installing and upgrading on Openshift Container Platform 4.1. For Fuse 7.4, the Apicurito Operator is available in the OperatorHub. Follow the steps below to install the operator.

Prerequisites

- You have access to OpenShift Server.

- You have configured authentication to the Red Hat Container Registry.

Procedure

1. Start the OpenShift 4.x Server.
2. Navigate to the OpenShift console in your browser. Log in to the console with your credentials.
3. Click **Catalog** and then Click **OperatorHub**.
4. Enter **Apicurito** in the search field window and press **Enter** key. You can see the Apicurito Operator in the right hand side panel.
5. Click **Apicurito Operator**. The Apicurito Operator install window is displayed.
6. Click **Install**. The **Create Operator Subscription** form is displayed. The form has options for:
    - Installation mode
    - Update Channel
    - Approval Strategy.
      Select as per you preferences or you can keep the default values.
7. Click **Subscribe**. The Apicurito Operator is now installed in the selected namespace.
8. To verify, click **Catalog** and then click **Installed Operator**. You can see the Apicurito Operator in the list.
9. Click **Apicurito Operator** in the Name column. Click **Create New** under **Provided APIs**. A new Custom Resource Definition (CRD) is created with the default values. Following fields are supported as part of the CR:
    - size: how many pods your the apicurito operator will have.
    - image: the apicurito image. Changing this image in an existing installation will trigger an upgrade of the operator.
10. Click **Create**. This will create a new **apicurito-service**.

2.4. INSTALLING FUSE IMAGESTREAMS AND TEMPLATES ON THE OPENSHIFT 3.X SERVER

After you configure authentication to the Red Hat container registry, import and use the Red Hat Fuse on OpenShift image streams and templates.

Procedure

1. Start the OpenShift Server.
2. Log in to the OpenShift Server as an administrator.

    ```
    oc login -u system:admin
    ```

3. Verify that you are using the project for which you created a docker-registry secret.

```
oc project openshift
```

4. Install the Fuse on OpenShift image streams.

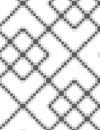
```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003

oc create -n openshift -f ${BASEURL}/fis-image-streams.json
```

5. Install the quickstart templates:

```
for template in eap-camel-amq-template.json \
 eap-camel-cdi-template.json \
 eap-camel-cxf-jaxrs-template.json \
 eap-camel-cxf-jaxws-template.json \
 eap-camel-jpa-template.json \
 karaf-camel-amq-template.json \
 karaf-camel-log-template.json \
 karaf-camel-rest-sql-template.json \
 karaf-cxf-rest-template.json \
 spring-boot-camel-amq-template.json \
 spring-boot-camel-config-template.json \
 spring-boot-camel-drools-template.json \
 spring-boot-camel-infinispan-template.json \
 spring-boot-camel-rest-sql-template.json \
 spring-boot-camel-template.json \
 spring-boot-camel-xa-template.json \
 spring-boot-camel-xml-template.json \
 spring-boot-cxf-jaxrs-template.json \
 spring-boot-cxf-jaxws-template.json ;
 do
 oc create -n openshift -f \
 https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003/quickstarts/${template}
 done
```

6. Install the templates for the Fuse Console.

```
oc create -n openshift -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003/fis-console-cluster-template.json
oc create -n openshift -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003/fis-console-namespace-template.json
```



NOTE

For information on deploying the Fuse Console, see Set up Fuse Console on OpenShift.

7. Install the Apicurito template:

```
oc create -n openshift -f ${BASEURL}/fuse-apicurito.yml
```

8. *(Optional)* View the installed Fuse on OpenShift images and templates:

```
oc get template -n openshift
```

CHAPTER 3. INSTALLING FUSE ON OPENSHIFT AS A NON-ADMIN USER

You can start using Fuse on OpenShift by creating an application and deploying it to OpenShift. First you need to install Fuse on OpenShift images and templates.

3.1. INSTALLING FUSE ON OPENSHIFT IMAGES AND TEMPLATES AS A NON-ADMIN USER

Prerequisites

- You have access to OpenShift server. It can be either virtual OpenShift server by CDK or remote OpenShift server.
- You have configured authentication to the Red Hat Container Registry.

For more information see:

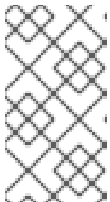
- Configuring Red Hat Container Registry Authentication .
- Red Hat CDK 3.9 Getting Started Guide

Procedure

1. In preparation for building and deploying the Fuse on OpenShift project, log in to the OpenShift Server as follows.

```
oc login -u developer -p developer https://OPENSHIFT_IP_ADDR:8443
```

   Where, **OPENSHIFT_IP_ADDR** is a placeholder for the OpenShift server's IP address as this IP address is not always the same.



   NOTE

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

2. Create a new project namespace called **test** (assuming it does not already exist).

```
oc new-project test
```

   If the **test** project namespace already exists, switch to it.

```
oc project test
```

3. Install the Fuse on OpenShift image streams:

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003
```

```
oc create -n test -f ${BASEURL}/fis-image-streams.json
```

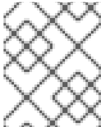
The command output displays the Fuse image streams that are now available in your Fuse on OpenShift project.

4. Install the quickstart templates.

```
for template in eap-camel-amq-template.json \
 eap-camel-cdi-template.json \
 eap-camel-cxf-jaxrs-template.json \
 eap-camel-cxf-jaxws-template.json \
 eap-camel-jpa-template.json \
 karaf-camel-amq-template.json \
 karaf-camel-log-template.json \
 karaf-camel-rest-sql-template.json \
 karaf-cxf-rest-template.json \
 spring-boot-camel-amq-template.json \
 spring-boot-camel-config-template.json \
 spring-boot-camel-drools-template.json \
 spring-boot-camel-infinispan-template.json \
 spring-boot-camel-rest-sql-template.json \
 spring-boot-camel-template.json \
 spring-boot-camel-xa-template.json \
 spring-boot-camel-xml-template.json \
 spring-boot-cxf-jaxrs-template.json \
 spring-boot-cxf-jaxws-template.json ;
 do
 oc create -n test -f \
 ${BASEURL}/quickstarts/${template}
 done
```

5. Install the templates for the Fuse Console.

```
oc create -n test -f ${BASEURL}/fis-console-cluster-template.json
oc create -n test -f ${BASEURL}/fis-console-namespace-template.json
```



NOTE

For information on deploying the Fuse Console, see Set up Fuse Console on OpenShift.

6. (Optional) View the installed Fuse on OpenShift images and templates.

```
oc get template -n test
```

7. In your browser, navigate to the OpenShift console:

    a. Use https://OPENSHIFT_IP_ADDR:8443 and replace **OPENSHIFT_IP_ADDR** with your OpenShift server's IP address.

    b. Log in to the OpenShift console with your credentials (for example, with username **developer** and password **developer**).

CHAPTER 4. GETTING STARTED FOR DEVELOPERS

4.1. PREPARING DEVELOPMENT ENVIRONMENT

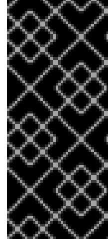
The fundamental requirement for developing and testing Fuse on OpenShift projects is having access to an OpenShift Server. You have the following basic alternatives:

- Install Red Hat CDK
- Getting Remote Access to an Existing OpenShift Server

4.1.1. Installing Container Development Kit (CDK) on Your Local Machine

As a developer, if you want to get started quickly, the most practical alternative is to install Red Hat CDK on your local machine. Using 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 CDK consists of the following key components:

- A virtual machine (libvirt, VirtualBox, or Hyper-V)
- Minishift to start and manage the Container Development Environment



IMPORTANT

Red Hat CDK is intended for *development purposes only*. It is not intended for other purposes, such as production environments, and may not address known security vulnerabilities. For full support of running mission-critical applications inside of docker-formatted containers, you need an active RHEL 7 or RHEL Atomic subscription. For more details, see Support for Red Hat Container Development Kit (CDK) .

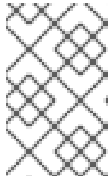
Prerequisites

- Java Version
  On your developer machine, make sure you have installed a Java version that is supported by Fuse 7.4. For details of the supported Java versions, see Supported Configurations.

Procedure

To install the CDK on your local machine:

1. For Fuse on OpenShift, we recommend that you install version 3.9 of CDK. Detailed instructions for installing and using CDK 3.9 are provided in the Red Hat CDK 3.9 Getting Started Guide .
2. Configure your OpenShift credentials to gain access to the Red Hat container registry by following the instructions in Configuring Red Hat Container Registry authentication .
3. Install the Fuse on OpenShift images and templates manually as described in Chapter 2, *Getting Started for Administrators*.



NOTE

Your version of CDK might have Fuse on OpenShift images and templates pre-installed. However, you must install (or update) the Fuse on OpenShift images and templates after you configure your OpenShift credentials.

4. Before you proceed with the examples in this chapter, you should read and thoroughly understand the contents of the Red Hat CDK 3.9 Getting Started Guide .

4.1.2. Getting 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 Fuse on OpenShift:

- The server machines must be running a supported version of OpenShift Container Platform (as documented in the Supported Configurations page). The examples in this guide have been tested against version 3.11.
- Ask the OpenShift administrator to install the latest Fuse on OpenShift container base images and the Fuse on OpenShift 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.

4.1.3. Installing Client-Side Tools

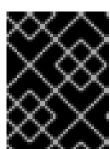
We recommend that you have the following tools installed on your developer machine:

- Apache Maven 3.6.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 3.6.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 Fuse on OpenShift projects. Download the appropriate package from the Git Downloads page.
- OpenShift client: If you are using CDK, you can add the **oc** binary to your PATH using **minishift oc-env** which displays the command you need to type into your shell (the output of **oc-env** will differ depending on OS and shell type):

  ```
  $ minishift oc-env
  export PATH="/Users/john/.minishift/cache/oc/v1.5.0:$PATH"
  # Run this command to configure your shell:
  # eval $(minishift oc-env)
  ```

  For more details, see Using the OpenShift Client Binary in CDK 3.9 *Getting Started Guide*.

  If you are not using 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). For information about specific binary installations for your operating system, see the Docker installation site.
  For more details, see Reusing the docker Daemon in CDK 3.9 *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.

Additional Resources

(Optional) Red Hat JBoss CodeReady Studio: *Red Hat JBoss CodeReady Studio* is an Eclipse-based development environment that includes support for developing Fuse on OpenShift applications. For details about how to install this development environment, see Install Red Hat JBoss CodeReady Studio.

4.1.4. Configuring Maven Repositories

Configure the Maven repositories, which hold the archetypes and artifacts that you will need for building an Fuse on OpenShift project on your local machine.

Procedure

1. Open your Maven **settings.xml** file, which is usually located in **~/.m2/settings.xml** (on Linux or macOS) or **Documents and Settings\<USER_NAME>\.m2\settings.xml** (on Windows).

2. Add the following Maven repositories.

   - 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.

4.2. CREATING AND DEPLOYING APPLICATIONS ON FUSE ON OPENSHIFT

You can start using Fuse on OpenShift 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 characterized 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 3.5 Developer Guide*.

S2I source workflow

S2I with build input from a *Git source*. This workflow is characterized by the fact that the build is executed entirely on the OpenShift server. For more details, see Git Source from the *OpenShift 3.5 Developer Guide*.

4.2.1. Creating and Deploying a Project Using the S2I Binary Workflow

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

Procedure

1. Create a new Fuse on OpenShift project using a Maven archetype. For this example, we use an archetype that creates a sample Spring Boot Camel project. Open a new shell 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/2.2.0.fuse-740017-redhat-00003/archetypes-catalog-2.2.0.fuse-740017-redhat-
00003-archetype-catalog.xml \
  -DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
  -DarchetypeArtifactId=spring-boot-camel-xml-archetype \
  -DarchetypeVersion=2.2.0.fuse-740017-redhat-00003
```

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': : fuse74-spring-boot
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.example.fis: :
[INFO] Using property: spring-boot-version = 1.5.17.RELEASE
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse74-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
spring-boot-version: 1.5.17.RELEASE
 Y: :
```

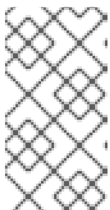
When prompted, enter **org.example.fis** for the **groupId** value and **fuse74-spring-boot** for the **artifactId** value. Accept the defaults for the remaining fields.

2. If the previous command exited with the **BUILD SUCCESS** status, you should now have a new Fuse on OpenShift project under the **fuse74-spring-boot** subdirectory. You can inspect the XML DSL code in the **fuse74-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 Fuse on OpenShift project, log in to the OpenShift Server as follows.

```
oc login -u developer -p developer https://OPENSHIFT_IP_ADDR:8443
```

Where, **OPENSHIFT_IP_ADDR** is a placeholder for the OpenShift server's IP address as this IP address is not always the same.



NOTE

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

4. Run the following command to ensure that Fuse on OpenShift images and templates are already installed and you can access them.

```
oc get template -n openshift
```

If the images and templates are not pre-installed, or if the provided versions are out of date, install (or update) the Fuse on OpenShift images and templates manually. For more information on how to install Fuse on OpenShift images see Chapter 2, *Getting Started for Administrators*.

5. 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.

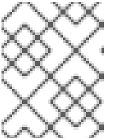
```
oc project test
```

6. You are now ready to build and deploy the **fuse74-spring-boot** project. Assuming you are still logged into OpenShift, change to the directory of the **fuse74-spring-boot** project, and then build and deploy the project, as follows.

```
cd fuse74-spring-boot
mvn fabric8:deploy -Popenshift
```

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 fuse74-spring-boot
[INFO] Created Service: target/fabric8/applyJson/test/service-fuse74-spring-boot.json
[INFO] Using project: test
[INFO] Creating a DeploymentConfig from openshift.yml namespace test name fuse74-
spring-boot
[INFO] Created DeploymentConfig: target/fabric8/applyJson/test/deploymentconfig-fuse74-
spring-boot.json
[INFO] Creating Route test:fuse74-spring-boot host: null
[INFO] F8: HINT: Use the command `oc get pods -w` to watch your pods start up
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 05:38 min
[INFO] Finished at: 2019-02-22T12:08:11+01:00
[INFO] Final Memory: 63M/272M
[INFO] ------------------------------------------------------------------------
```



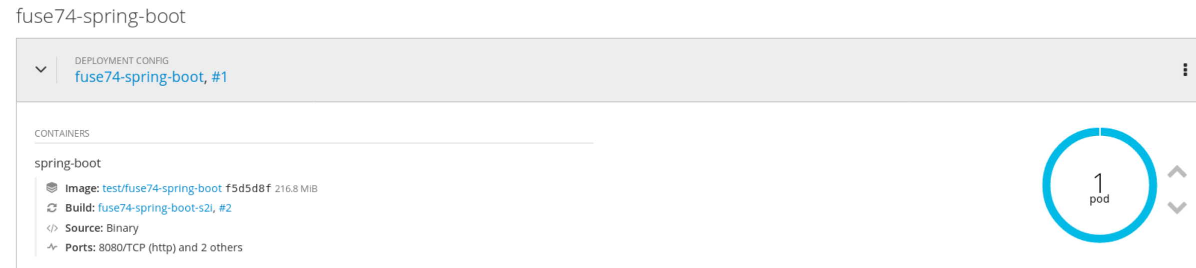
NOTE

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

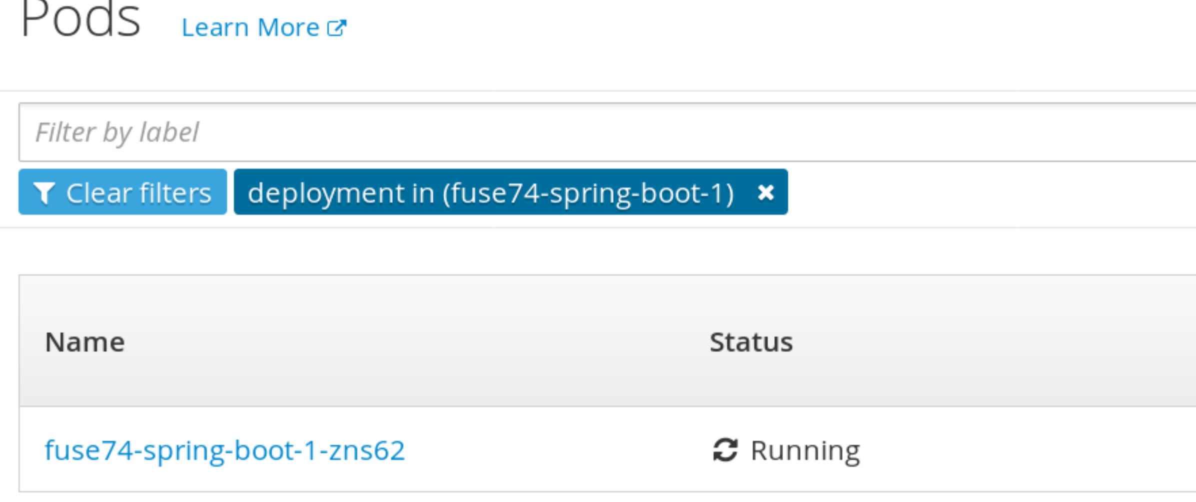
7. Navigate to the OpenShift console in your browser and log in to the console with your credentials (for example, with username **developer** and password, **developer**).

8. In the OpenShift console, scroll down to find the **test** project namespace. Click the **test** project to open the **test** project namespace. The **Overview** tab of the **test** project opens, showing the **fuse74-spring-boot** application.

9. Click the arrow on the left of the **fuse74-spring-boot** deployment to expand and view the details of this deployment, as shown.



10. Click in the center of the pod icon (blue circle) to view the list of pods for **fuse74-spring-boot**.



11. Click on the pod **Name** (in this example, **fuse74-spring-boot-1-kxdjm**) to view the details of the running pod.

Pods » fuse74-spring-boot-1-zns62

fuse74-spring-boot-1-zns62 created 6 minutes ago

app fuse74-spring-boot deployment fuse74-spring-boot-1 deploymentconfig

Details Environment Logs Terminal Events

Status

Status: Running
Deployment: fuse74-spring-boot, #1
IP: 172.17.0.6
Node: localhost (192.168.122.218)
Restart Policy: Always

Container spring-boot

State: Running since Aug 16, 2019 7:11:28 PM
Ready: true
Restart Count: 0

12. 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
```

13. Click **Overview** on the left-hand navigation bar to return to the applications overview 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 fuse74-spring-boot-1?**, click **Scale Down**.

14. *(Optional)* If you are using CDK, you can shut down the virtual OpenShift Server completely by returning to the shell prompt and entering the following command:

```
minishift stop
```

4.2.2. Undeploying and Redeploying the Project

You can undeploy or redeploy your projects, as follows:

Procedure

- To undeploy the project, enter the command:

```
mvn fabric8:undeploy
```

- To redeploy the project, enter the commands:

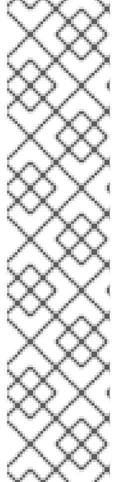
```
mvn fabric8:undeploy
mvn fabric8:deploy -Popenshift
```

4.2.3. Set up Fuse Console on OpenShift

In OpenShift, you can access the Fuse Console in two ways:

- From a specific pod so that you can monitor that single running Fuse container.
- By adding the *centralized* Fuse Console catalog item to your project so that you can monitor all the running Fuse containers in your project.

You can deploy the Fuse Console either from the OpenShift Console or from the command line.



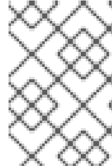
NOTE

- On OpenShift 4, if you want to manage Fuse 7.4 services with the Fuse Console, you must install the community version (Hawtio) as described in the Fuse 7.4 release notes.
- Security and user management for the Fuse Console is handled by OpenShift.
- The Fuse Console templates configure end-to-end encryption by default so that your Fuse Console requests are secured end-to-end, from the browser to the in-cluster services.
- Role-based access control (for users accessing the Fuse Console after it is deployed) is not yet available for Fuse on OpenShift.

Prerequisites

- Install the Fuse on OpenShift image streams and the templates for the Fuse Console as described in Fuse on OpenShift Guide.
- If you want to deploy the Fuse Console in cluster mode on the OpenShift Container Platform environment, you need the cluster admin role and the cluster mode template. Run the following command:

```
oc adm policy add-cluster-role-to-user cluster-admin system:serviceaccount:openshift-infra:template-instance-controller
```



NOTE

The cluster mode template is only available, by default, on the latest version of the OpenShift Container Platform. It is not provided with the OpenShift Online default catalog.

4.2.3.1. Monitoring a single Fuse pod from the Fuse Console

You can open the Fuse Console for a Fuse pod running on OpenShift:

1. From the **Applications → Pods** view in your OpenShift project, click on the pod name to view the details of the running Fuse pod. On the right-hand side of this page, you see a summary of the container template:

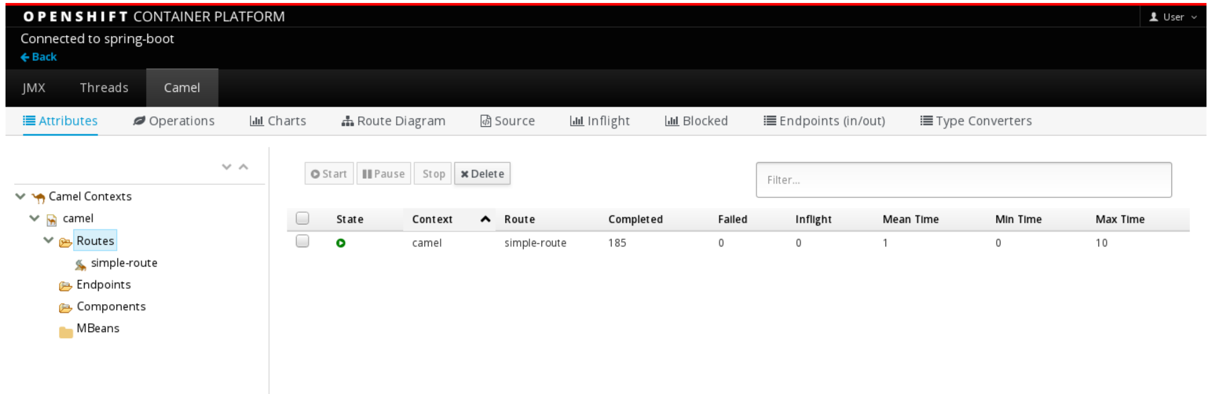
   Template

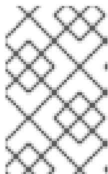
   Containers

   CONTAINER: SPRING-BOOT

   Image: test/fuse70-spring-boot eda527f 193.1 MiB
   Build: fuse70-spring-boot-s2i, #2
   Source: Binary
   Ports: 8080/TCP (http), 8778/TCP (jolokia), 9779/TCP (prometheus)
   Mount: default-token-p4zsn → /var/run/secrets/kubernetes.io/serviceaccount read-only
   CPU: 200 millicores to 1 core
   Readiness Probe: GET /health on port 8081 (HTTP) 10s delay, 1s timeout
   Liveness Probe: GET /health on port 8081 (HTTP) 180s delay, 1s timeout
   Open Java Console

2. From this view, click on the **Open Java Console** link to open the Fuse Console.





NOTE

In order to configure OpenShift to display a link to Fuse Console in the pod view, the pod running a Fuse on OpenShift image must declare a TCP port within a name attribute set to **jolokia**:

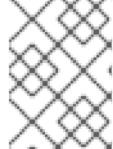
```
{
  "kind": "Pod",
  [...]
  "spec": {
    "containers": [
      {
        [...]
        "ports": [
          {
            "name": "jolokia",
            "containerPort": 8778,
            "protocol": "TCP"
          }
```

4.2.3.2. Deploying the Fuse Console from the OpenShift Console

To deploy the Fuse Console on your OpenShift cluster from the OpenShift Console, follow these steps.

Procedure

1. In the OpenShift console, open an existing project or create a new project.
2. Add the Fuse Console to your OpenShift project:
   a. Select **Add to Project** → **Browse Catalog**.
      The **Select an item to add to the current project** page opens.
   b. In the **Search** field, type **Fuse Console**.
      The **Red Hat Fuse 7.x Console** and **Red Hat Fuse 7.x Console (cluster)** items should appear as the search result.



NOTE

If the **Red Hat Fuse Console** items do not appear as the search result, or if the items that appear are not the latest version, you can install the Fuse Console templates manually as described in the "Prepare the OpenShift server" section of the Fuse on OpenShift Guide.

a. Click one of the **Red Hat Fuse Console** items:
   - **Red Hat Fuse 7.x Console** - This version of the Fuse Console discovers and connects to Fuse applications deployed in the current OpenShift project.
   - **Red Hat Fuse 7.x Console (cluster)** - This version of the Fuse Console can discover and connect to Fuse applications deployed across multiple projects on the OpenShift cluster.

b. In the **Red Hat Fuse Console** wizard, click **Next**. The **Configuration** page of the wizard opens. Optionally, you can change the default values of the configuration parameters.
   1. Click **Create**.
      The **Results** page of the wizard indicates that the Red Hat Fuse Console has been created.
   2. Click the **Continue to the project overview** link to verify that the Fuse Console application is added to the project.
   3. To open the Fuse Console, click the provided URL link and then log in.

An **Authorize Access** page opens in the browser listing the required permissions.

4. Click **Allow selected permissions**.
   The Fuse Console opens in the browser and shows the Fuse pods running in the project.

5. Click **Connect** for the application that you want to view.
   A new browser window opens showing the application in the Fuse Console.

4.2.3.3. Deploying the Fuse Console from the command line

Table 4.1, "Fuse Console templates" describes the two OpenShift templates that you can use to access the Fuse Console from the command line, depending on the type of Fuse application deployment.

Table 4.1. Fuse Console templates

Type	Description
cluster	Use an OAuth client that requires the cluster-admin role to be created. The Fuse Console can discover and connect to Fuse applications deployed across multiple namespaces or projects.
namespace	Use a service account as OAuth client, which only requires the admin role in a project to be created. This restricts the Fuse Console access to this single project, and as such acts as a single tenant deployment.

Optionally, you can view a list of the template parameters by running the following command:

```
oc process --parameters -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003/fis-console-namespace-template.json
```

Procedure

To deploy the Fuse Console from the command line:

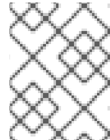
1. Create a new application based on a Fuse Console template by running one of the following commands (where **myproject** is the name of your project):

   - For the Fuse Console **cluster** template, where **myhost** is the hostname to access the Fuse Console:

     ```
     oc new-app -n myproject -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003/fis-console-cluster-template.json -p ROUTE_HOSTNAME=myhost
     ```

   - For the Fuse Console **namespace** template:

     ```
     oc new-app -n myproject -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003/fis-console-namespace-template.json
     ```



NOTE

You can omit the route_hostname parameter for the **namespace** template because OpenShift automatically generates one.

2. Obtain the status and the URL of your Fuse Console deployment by running this command:

```
oc status
```

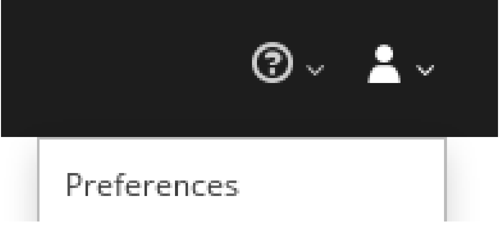
3. To access the Fuse Console from a browser, use the provided URL (for example, https://fuse-console.192.168.64.12.nip.io).

4.2.3.4. Ensuring that data displays correctly in the Fuse Console

If the display of the queues and connections in the Fuse Console is missing queues, missing connections, or displaying inconsistent icons, adjust the Jolokia collection size parameter that specifies the maximum number of elements in an array that Jolokia marshals in a response.

Procedure

1. In the upper right corner of the Fuse Console, click the user icon and then click **Preferences**.



2. Increase the value of the **Maximum collection size** option (the default is 50,000).

3. Click **Close**.

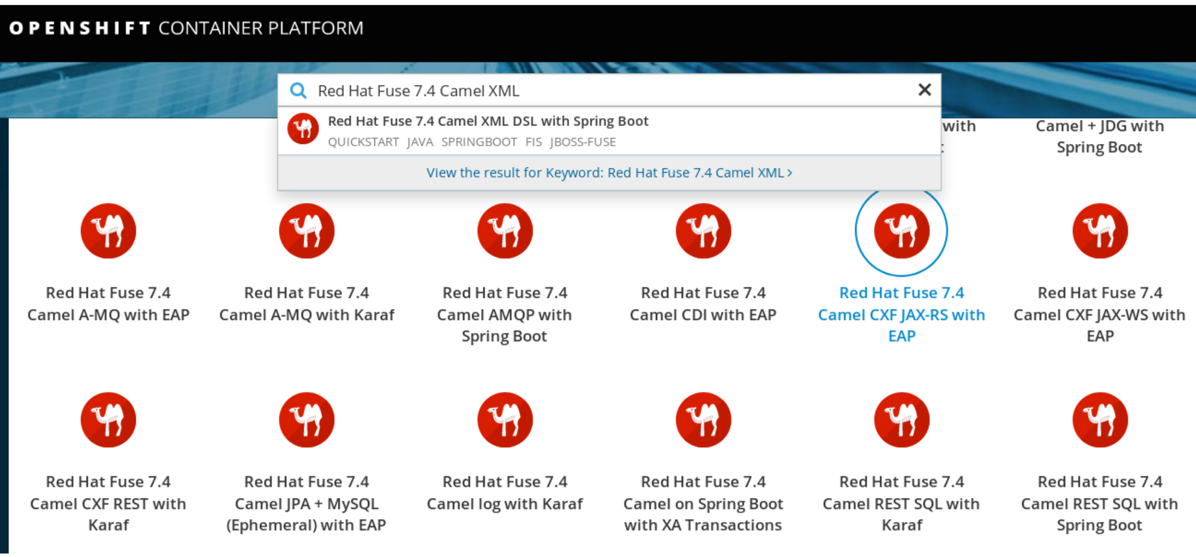
4.2.4. Creating and Deploying a Project Using the S2I Source Workflow

In this section, you will use the OpenShift S2I source workflow to build and deploy a Fuse on OpenShift 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.

Procedure

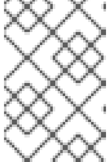
1. Navigate to the OpenShift console in your browser (https://OPENSHIFT_IP_ADDR:8443, replace **OPENSHIFT_IP_ADDR** with the IP address that was displayed in the case of CDK) and log in to the console with your credentials (for example, with username **developer** and password, **developer**).

2. In the Catalog search field, enter **Red Hat Fuse 7.4 Camel XML** as the search string and select the **Red Hat Fuse 7.4 Camel XML DSL with Spring Boot**template.



3. The **Information** step of the template wizard opens. Click **Next**.

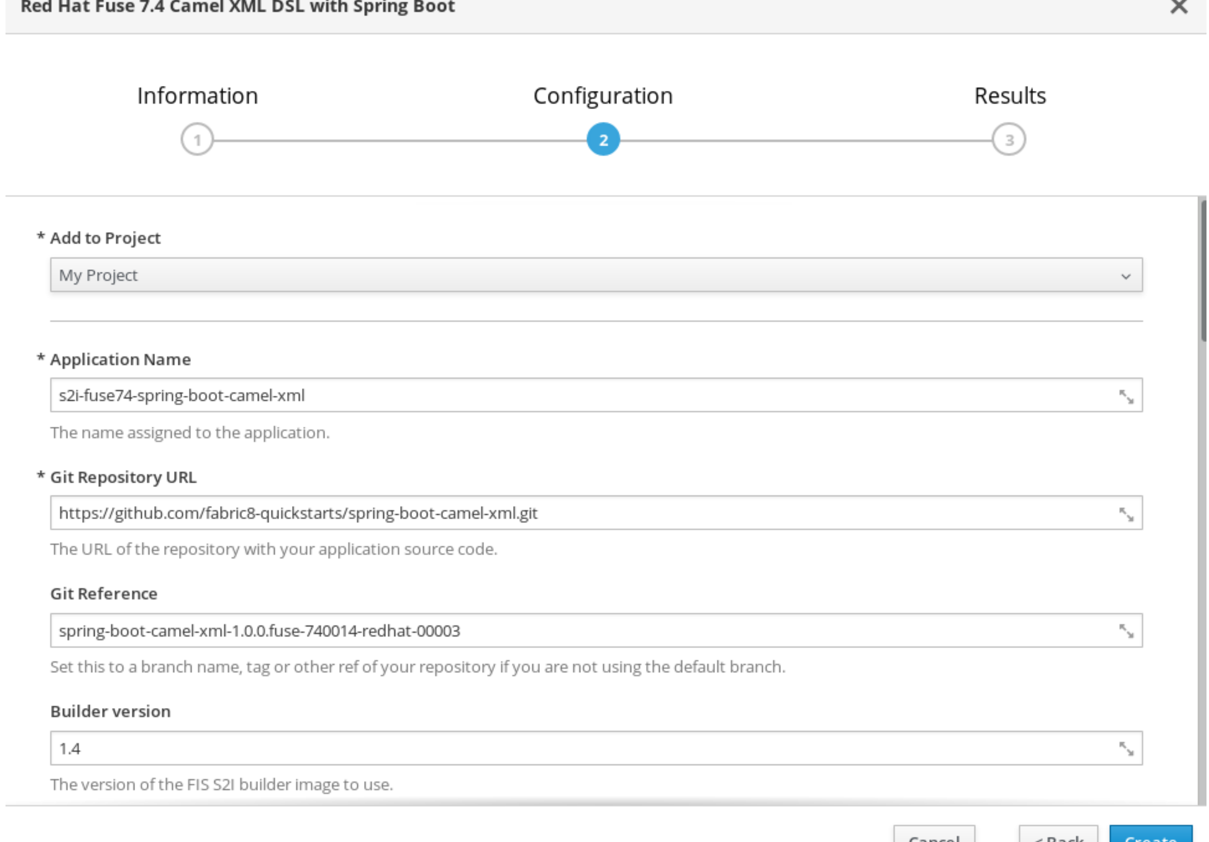
4. The **Configuration** step of the template wizard opens, as shown. From the **Add to Project** dropdown, select **My Project**.

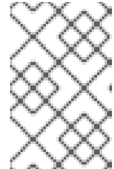


NOTE

Alternatively, if you prefer to create a new project for this example, select **Create Project** from the **Add to Project** dropdown. A **Project Name** field then appears for you to fill in the name of the new project.

5. You can accept the default values for the rest of the settings in the **Configuration** step. Click **Create**.





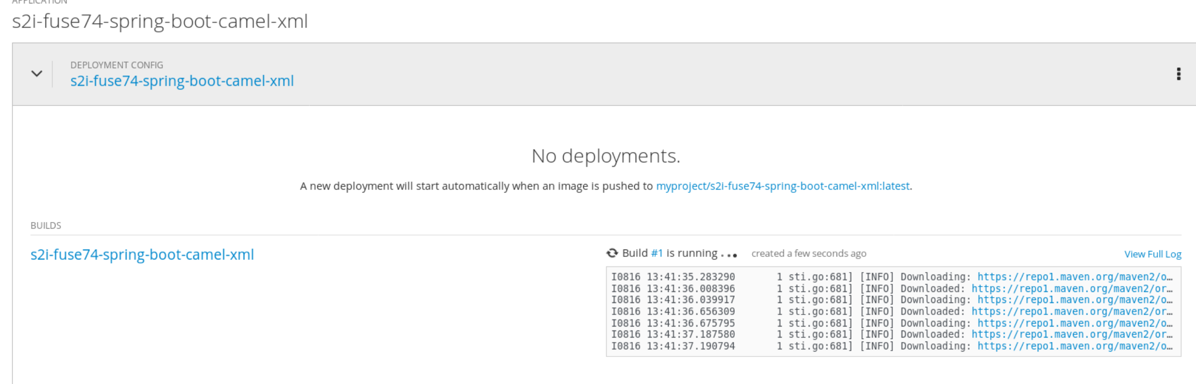
NOTE

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.

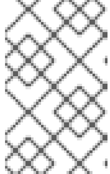
6. The **Results** step of the template wizard opens. Click **Close**.

7. In the right-hand **My Projects** pane, click **My Project**. The **Overview** tab of the **My Project** project opens, showing the **s2i-fuse74-spring-boot-camel-xml** application.

8. Click the arrow on the left of the **s2i-fuse74-spring-boot-camel-xml** deployment to expand and view the details of this deployment, as shown.



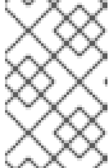
9. In this view, you can see the build log. If the build should fail for any reason, the build log can help you to diagnose the problem.



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.

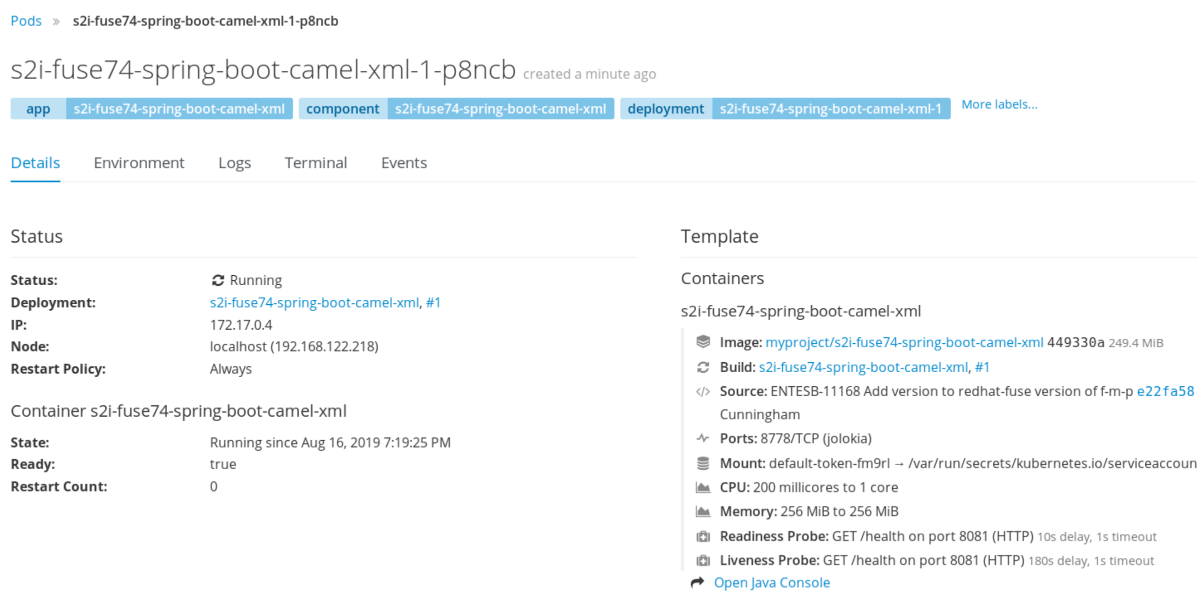
10. If the build completes successfully, the pod icon shows as a blue circle with **1 pod** running. Click in the centre of the pod icon (blue circle) to view the list of pods for **s2i-fuse74-spring-boot-camel-xml**.



NOTE

If multiple pods are running, you would see a list of running pods at this point. Otherwise (if there is just one pod), you get straight through to the details view of the running pod.

11. The pod details view opens. 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.



12. Click **Overview** on the left-hand navigation bar to return to the overview of the applications in the **My Project** 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-fuse74-spring-boot-camel-xml-1?**, click **Scale Down**.

13. *(Optional)* If you are using CDK, you can shut down the virtual OpenShift Server completely by returning to the shell prompt and entering the following command:

```
minishift stop
```

CHAPTER 5. DEVELOPING AN APPLICATION FOR THE SPRING BOOT IMAGE

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

5.1. CREATING A SPRING BOOT PROJECT USING MAVEN ARCHETYPE

To create a Spring Boot project using Maven archetypes follow these steps.

Procedure

1. Go to the appropriate directory on your system.
2. In a shell prompt, enter the following the **mvn** command to create a Spring Boot project.

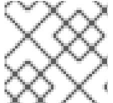
```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
  -
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-
catalog/2.2.0.fuse-740017-redhat-00003/archetypes-catalog-2.2.0.fuse-740017-redhat-
00003-archetype-catalog.xml \
  -DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
  -DarchetypeArtifactId=spring-boot-camel-xml-archetype \
  -DarchetypeVersion=2.2.0.fuse-740017-redhat-00003
```

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': : fuse74-spring-boot
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.example.fis: :
[INFO] Using property: spring-boot-version = 1.5.17.RELEASE
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse74-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
spring-boot-version: 1.5.17.RELEASE
 Y: :
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse74-spring-boot** for the **artifactId** value. Accept the defaults for the remaining fields.

3. Follow the instructions in the quickstart on how to build and deploy the example.



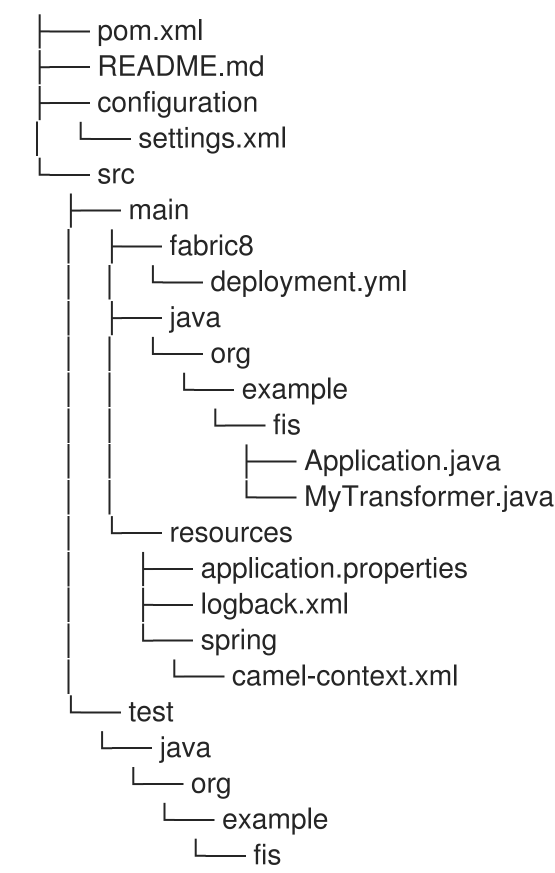
NOTE

For the full list of available Spring Boot archetypes, see Spring Boot Archetype Catalog.

5.2. 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
├── configuration
│   └── settings.xml
└── src
    ├── main
    │   ├── fabric8
    │   │   └── deployment.yml
    │   ├── java
    │   │   └── org
    │   │       └── example
    │   │           └── fis
    │   │               ├── Application.java
    │   │               └── MyTransformer.java
    │   └── resources
    │       ├── application.properties
    │       ├── logback.xml
    │       └── spring
    │           └── camel-context.xml
    └── test
        └── java
            └── org
                └── example
                    └── fis
```

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

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

```
#spring.main.sources=org.example.fos

logging.config=classpath:logback.xml

# 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

# lets use a different management port in case you need to listen to HTTP requests on 8080
management.port=8081
```

```
# disable all management endpoints except health
endpoints.enabled = false
endpoints.health.enabled = true
```

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 Blueprint file from the classpath that contains the Camel XML DSL
@ImportResource({"classpath:blueprint/camel-context.xml"})
```

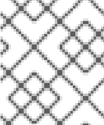
It must have a **main** method to run the Spring Boot application.

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.

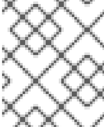


NOTE

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

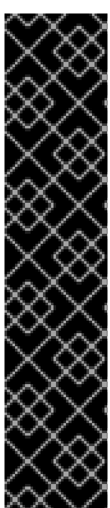
This file is not used part of Spring Boot application but it is used in all quickstarts to limit the resources such as CPU and memory usage.

5.3. SPRING BOOT ARCHETYPE CATALOG

The Spring Boot Archetype catalog includes the following examples.

Table 5.1. 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.
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-xa-archetype	Spring Boot, Camel and XA Transactions. This example demonstrates how to run a Camel Service on Spring Boot that supports XA transactions on two external transactional resources: a JMS resource (A-MQ) and a database (PostgreSQL). This quickstart requires the PostgreSQL database and the A-MQ broker have been deployed and running first, one simple way to run them is to use the templates provided in the Openshift service catalog
spring-boot-camel-xml-archetype	Demonstrates how to configure Camel routes in Spring Boot via a Blueprint 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 Boot based on a fabric8 Java base image. The quickstart uses Spring Boot to configure an application that includes a CXF JAXWS endpoint.



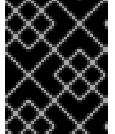
IMPORTANT

A Technology Preview quickstart is also available. The Spring Boot Camel XA Transactions quickstart demonstrates how to use Spring Boot to run a Camel service that supports XA transactions. This quickstart shows the use of two external transactional resources: a JMS (AMQ) broker and a database (PostgreSQL). You can find this quickstart here: https://github.com/jboss-fuse/spring-boot-camel-xa.

Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information, see Red Hat Technology Preview features support scope.

5.4. BOM FILE FOR SPRING BOOT

The purpose of a Maven Bill of Materials (BOM) file is to provide a curated set of Maven dependency versions that work well together, preventing you from having to define versions individually for every Maven artifact.

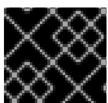


IMPORTANT

Ensure you are using the correct Fuse BOM based on the version of Spring Boot you are using (Spring Boot 1 or Spring Boot 2).

The Fuse BOM for Spring Boot offers the following advantages:

- Defines versions for Maven dependencies, so that you do not need to specify the version when you add a dependency to your POM.
- Defines a set of curated dependencies that are fully tested and supported for a specific version of Fuse.
- Simplifies upgrades of Fuse.



IMPORTANT

Only the set of dependencies defined by a Fuse BOM are supported by Red Hat.

5.4.1. Incorporate the BOM file

To incorporate a BOM file into your Maven project, specify a **dependencyManagement** element in your project's **pom.xml** file (or, possibly, in a parent POM file), as shown in the examples for both Spring Boot 2 and Spring Boot 1 below:

- Spring Boot 2 BOM
- Spring Boot 1 BOM

```
<?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 -->
    <fuse.version>7.4.0.fuse-sb2-740019-redhat-00005</fuse.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-springboot-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

```
<?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 -->
    <fuse.version>7.4.0.fuse-740036-redhat-00002</fuse.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-springboot-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

After specifying the BOM using the dependency management mechanism, it becomes possible to add Maven dependencies to your POM *without* specifying the version of the artifact. For example, to add a dependency for the **camel-hystrix** component, you would add the following XML fragment to the **dependencies** element in your POM:

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

Note how the Camel artifact ID is specified with the **-starter** suffix – that is, you specify the Camel Hystrix component as **camel-hystrix-starter**, not as **camel-hystrix**. The Camel starter components are packaged in a way that is optimized for the Spring Boot environment.

5.5. SPRING BOOT MAVEN PLUGIN

The Spring Boot Maven plugin is provided by Spring Boot and it is a developer utility for building and running a Spring Boot project:

- *Building* – create an executable Jar package for your Spring Boot application by entering the command **mvn package** in the project directory. The output of the build is placed in the **target/** subdirectory of your Maven project.
- *Running* – for convenience, you can run the newly-built application with the command, **mvn spring-boot:start**.

To incorporate the Spring Boot Maven plugin into your project POM file, add the plugin configuration to the **project/build/plugins** section of your **pom.xml** file, as shown in the following example:

```
<?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 -->
    <fuse.version>7.4.0.fuse-740036-redhat-00002</fuse.version>

  </properties>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>${fuse.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

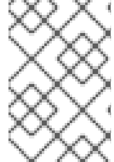
CHAPTER 6. APACHE CAMEL IN SPRING BOOT

6.1. INTRODUCTION TO CAMEL SPRING BOOT

The Camel 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>org.jboss.redhat-fuse</groupId>
   <artifactId>fuse-springboot-bom</artifactId>
   <version>${fuse.version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

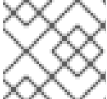


NOTE

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.

6.2. INTRODUCTION TO CAMEL SPRING BOOT STARTER

Apache Camel includes a 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");
    }
}
```

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

Camel Spring Boot now supports referring to bean by the id name in the configuration files (application.properties or yaml file) when you configure any of the Camel starter components. In the **src/main/resources/application.properties** (or yaml) file you can now easily configure the options on the Camel that refers to other beans by referring to the beans ID name. For example, the xslt component can refer to a custom bean using the bean ID as follows:

Refer to a custom bean by the id myExtensionFactory as follows:

```
camel.component.xslt.saxon-extension-functions=myExtensionFactory
```

Which you can then create using Spring Boot @Bean annotation as follows:

```
@Bean(name = "myExtensionFactory")
public ExtensionFunctionDefinition myExtensionFactory() {
    }
```

Or, in case of a Jackson ObjectMapper in the **camel-jackson** data-format:

```
camel.dataformat.json-jackson.object-mapper=myJacksonMapper
```

6.3. 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);
  }

}
```

6.4. 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");
      }

    };
  }

}
```

6.5. 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:

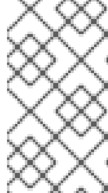
```
@Component
public class MyRouter extends RouteBuilder {
```

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

}
```

6.6. 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.

6.7. 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
```

6.8. 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
```

6.9. 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);
  }

}
```

6.10. SPRING TYPE CONVERSION API BRIDGE

Spring consist of 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**.

6.11. DISABLING TYPE CONVERSIONS FEATURES

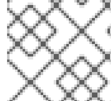
To disable registering type conversion features of Camel Spring Boot such as **TypeConverter** instance or Spring bridge, set the **camel.springboot.typeConversion** property to **false** as shown below:

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

6.12. 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>
```

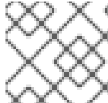
When using Spring XML files with <camelContext>, you can configure Camel in the Spring XML file as well as in the application.properties file. For example, to set a name on Camel and turn On the stream caching, add:

```
camel.springboot.name = MyCamel
camel.springboot.stream-caching-enabled=true
```

6.13. 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>
```

6.14. TESTING WITH CAMEL SPRING BOOT

In case on Camel running on Spring Boot, Spring Boot automatically embeds Camel and all its routes, which are annotated with **@Component**. When testing with Spring boot you use **@SpringBootTest** instead of **@ContextConfiguration** to specify which configuration class to use.

When you have multiple Camel routes in different RouteBuilder classes, Camel Spring Boot will include all these routes. Hence, when you wish to test routes from only one RouteBuilder class you can use the following patterns to include or exclude which RouteBuilders to enable:

- java-routes-include-pattern: Used for including RouteBuilder classes that match the pattern.
- java-routes-exclude-pattern: Used for excluding RouteBuilder classes that match the pattern. Exclude takes precedence over include.

You can specify these patterns in your unit test classes as properties to **@SpringBootTest** annotation, as shown below:

```
@RunWith(CamelSpringBootRunner.class)
@SpringBootTest(classes = {MyApplication.class);
   properties = {"camel.springboot.java-routes-include-pattern=**/Foo*"})
public class FooTest {
```

In the **FooTest** class, the include pattern is **/**Foo***, which represents an Ant style pattern. Here, the pattern starts with double asterisk, which matches with any leading package name. /**Foo*** means the class name must start with Foo, for example, FooRoute. You can run a test using following maven command:

```
mvn test -Dtest=FooTest
```

6.15. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 7. RUNNING A CAMEL SERVICE ON SPRING BOOT WITH XA TRANSACTIONS

The Spring Boot Camel XA transactions quickstart demonstrates how to run a Camel Service on Spring-Boot that supports XA transactions on two external transactional resources, a JMS resource (A-MQ) and a database (PostgreSQL). These external resources are provided by OpenShift which must be started before running this quickstart.

7.1. STATEFULSET RESOURCES

This quickstart uses OpenShift **StatefulSet** resources to guarantee uniqueness of transaction managers and require a PersistentVolume to store transaction logs. The application supports scaling on the StatefulSet resource. Each instance will have its own **in-process** recovery manager. A special controller guarantees that when the application is scaled down, all instances, that are terminated, complete all their work correctly without leaving pending transactions. The scale-down operation is rolled back by the controller if the recovery manager is not been able to flush all pending work before terminating. This quickstart uses Spring Boot Narayana recovery controller.

7.2. SPRING BOOT NARAYANA RECOVERY CONTROLLER

The Spring Boot Narayana recovery controller allows to gracefully handle the scaling down phase of a StatefulSet by cleaning pending transactions before termination. If a scaling down operation is executed and the pod is not clean after termination, the previous number of replicas is restored, hence effectively canceling the scaling down operation.

All pods of the StatefulSet require access to a shared volume that is used to store the termination status of each pod belonging to the StatefulSet. The pod-0 of the StatefulSet periodically checks the status and scale the StatefulSet to the right size if there's a mismatch.

In order for the recovery controller to work, edit permissions on the current namespace are required (role binding is included in the set of resources published to OpenShift). The recovery controller can be disabled using the **CLUSTER_RECOVERY_ENABLED** environment variable. In this case, no special permissions are required on the service account but any scale down operation may leave pending transactions on the terminated pod without notice.

7.3. CONFIGURING SPRING BOOT NARAYANA RECOVERY CONTROLLER

Following example shows how to configure Narayana to work on OpenShift with the recovery controller.

Procedure

1. This is a sample **application.properties** file. Replace the following options in the Kubernetes yaml descriptor.

```
# Cluster
cluster.nodename=1
cluster.base-dir=./target/tx

# Transaction Data
spring.jta.transaction-manager-id=${cluster.nodename}
spring.jta.log-dir=${cluster.base-dir}/store/${cluster.nodename}
```

```
# Narayana recovery settings
snowdrop.narayana.openshift.recovery.enabled=true
snowdrop.narayana.openshift.recovery.current-pod-name=${cluster.nodename}
# You must enable resource filtering in order to inject the Maven artifactId
snowdrop.narayana.openshift.recovery.statefulset=${project.artifactId}
snowdrop.narayana.openshift.recovery.status-dir=${cluster.base-dir}/status
```

2. You need a shared volume to store both transactions and information related to termination. It can be mounted in the StatefulSet yaml descriptor as follows.

```
apiVersion: apps/v1beta1
kind: StatefulSet
#...
spec:
#...
  template:
#...
    spec:
      containers:
      - env:
        - name: CLUSTER_BASE_DIR
          value: /var/transaction/data
          # Override CLUSTER_NODENAME with Kubernetes Downward API (to use `pod-0`,
`pod-1` etc. as tx manager id)
        - name: CLUSTER_NODENAME
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.name
#...
        volumeMounts:
        - mountPath: /var/transaction/data
          name: the-name-of-the-shared-volume
#...
```

Camel Extension for Spring Boot Narayana Recovery Controller

If Camel is found in the Spring Boot application context, the Camel context is automatically stopped before flushing all pending transactions.

7.4. RUNNING CAMEL SPRING BOOT XA QUICKSTART ON OPENSHIFT

This procedure shows how to run the quickstart on a running single node OpenShift cluster.

Procedure

1. Download Camel Spring Boot XA project.

```
git clone https://github.com/jboss-fuse/spring-boot-camel-xa
```

2. Navigate to **spring-boot-camel-xa** directory and run following command.

```
mvn clean install
```

3. Log in to the OpenShift Server.

```
oc login -u developer -p developer
```

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

```
oc new-project test
```

If the **test** project namespace already exists, switch to it.

```
oc project test
```

5. Install dependencies.

   - From the OpenShift catalog, install **postgresql** using username as **theuser** and password as **Thepassword1!**.
   - From the OpenShift catalog, install the **A-MQ** broker using username as **theuser** and password as **Thepassword1!**.

6. Change the **Postgresql** database to accept prepared statements.

```
oc env dc/postgresql POSTGRESQL_MAX_PREPARED_TRANSACTIONS=100
```

7. Create a persistent volume claim for the transaction log.

```
oc create -f persistent-volume-claim.yml
```

8. Build and deploy your quickstart.

```
mvn fabric8:deploy -P openshift
```

9. Scale it up to the desired number of replicas.

```
oc scale statefulset spring-boot-camel-xa --replicas 3
```

Note: The pod name is used as transaction manager id (spring.jta.transaction-manager-id property). The current implementation also limits the length of transaction manager ids. So please note that:

   - The name of the StatefulSet is an identifier for the transaction system, so it must not be changed.
   - You should name the StatefulSet so that all of its pod names have length lower than or equal to 23 characters. Pod names are created by OpenShift using the convention: <statefulset-name>-0, <statefulset-name>-1 and so on. Narayana does its best to avoid having multiple recovery managers with the same id, so when the pod name is longer than the limit, the last 23 bytes are taken as transaction manager id (after stripping some characters like -).

10. Once the quickstart is running, get the base service URL using the following command.

```
NARAYANA_HOST=$(oc get route spring-boot-camel-xa -o jsonpath={.spec.host})
```

7.5. TESTING SUCCESSFUL XA TRANSACTIONS

Following workflow shows how to test the successful XA transactions.

Procedure

1. Get the list of messages in the audit_log table.

   ```
   curl -w "\n" http://$NARAYANA_HOST/api/
   ```

2. The list is empty at the beginning. Now you can put the first element.

   ```
   curl -w "\n" -X POST http://$NARAYANA_HOST/api/?entry=hello
   ```

   After waiting for some time get the new list.

   ```
   curl -w "\n" http://$NARAYANA_HOST/api/
   ```

3. The new list contains two messages, **hello** and **hello-ok**. The **hello-ok** confirms that the message has been sent to a outgoing queue and then logged. You can add multiple messages and see the logs.

7.6. TESTING FAILED XA TRANSACTIONS

Following workflow shows how to test the failed XA transactions.

Procedure

1. Send a message named **fail**.

   ```
   curl -w "\n" -X POST http://$NARAYANA_HOST/api/?entry=fail
   ```

2. After waiting for some time get the new list.

   ```
   curl -w "\n" http://$NARAYANA_HOST/api/
   ```

3. This message produces an exception at the end of the route, so that the transaction is always rolled back. You should not find any trace of the message in the audit_log table.

CHAPTER 8. INTEGRATING A CAMEL APPLICATION WITH THE A-MQ BROKER

This tutorial shows how to deploy a quickstart using the A-MQ image.

8.1. BUILDING AND DEPLOYING A SPRING BOOT CAMEL A-MQ QUICKSTART

This example requires a JBoss A-MQ 6 image and deployment template. If you are using CDK 3.1.1+, JBoss A-MQ 6 images and templates should be already installed in the **openshift** namespace by default.

Prerequisites

- Ensure that OpenShift is running correctly and the Fuse image streams are already installed in OpenShift. See Getting Started for Administrators.
- Ensure that Maven Repositories are configured for fuse, see Configuring Maven Repositories.

Procedure

1. Get ready to build and deploy the quickstart:

   a. Log in to OpenShift as a developer.

   ```
   oc login -u developer -p developer
   ```

   b. Create a new project **amq-quickstart**.

   ```
   oc new-project amq-quickstart
   ```

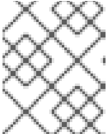
   c. Determine the version of the A-MQ 6 images and templates installed.

   ```
   $ oc get template -n openshift
   ```

   You should be able to find a template named **amqXX-basic**, where **XX** is the version of A-MQ installed in Openshift.

2. Deploy the A-MQ 6 image in the **amq-quickstart** namespace (replace **XX** with the actual version of A-MQ found in previous step).

   ```
   $ oc process openshift//amqXX-basic -p APPLICATION_NAME=broker -p
   MQ_USERNAME=admin -p MQ_PASSWORD=admin -p MQ_QUEUES=test -p
   MQ_PROTOCOL=amqp -n amq-quickstart | oc create -f -
   ```

   NOTE

   This **oc** command could fail, if you use an older version of **oc**. This syntax works with **oc** versions 3.5.x (based on Kubernetes 1.5.x).

3. Add a user role that is needed for discovery of mesh endpoints (through Kubernetes REST API agent).

```
$ oc policy add-role-to-user view system:serviceaccount:amq-quickstart:default
```

4. Create the quickstart project using the Maven workflow.

```
$ mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
-
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-
catalog/2.2.0.fuse-740017-redhat-00003/archetypes-catalog-2.2.0.fuse-740017-redhat-
00003-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-amq-archetype \
-DarchetypeVersion=2.2.0.fuse-740017-redhat-00003
```

5. 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': : fuse74-spring-boot-camel-amq
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.example.fis: :
[INFO] Using property: spring-boot-version = 1.5.17.RELEASE
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse74-spring-boot-camel-amq
version: 1.0-SNAPSHOT
package: org.example.fis
spring-boot-version: 1.5.17.RELEASE
 Y: :
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse74-spring-boot-camel-amq** for the **artifactId** value. Accept the defaults for the remaining fields.

6. Navigate to the quickstart directory **fuse74-spring-boot-camel-amq**.

```
$ cd fuse74-spring-boot-camel-amq
```

7. Customize the client credentials for logging on to the broker, by setting the **ACTIVEMQ_BROKER_USERNAME** and **ACTIVEMQ_BROKER_PASSWORD** environment variables. In the **fuse74-spring-boot-camel-amq** project, edit the **src/main/fabric8/deployment.yml** file, as follows:

```
spec:
  template:
    spec:
      containers:
        -
          resources:
            requests:
              cpu: "0.2"
#             memory: 256Mi
            limits:
              cpu: "1.0"
#             memory: 256Mi
          env:
          - name: AMQP_HOST
```

```
          value: broker-amq-amqp
        - name: SPRING_APPLICATION_JSON
          value: '{"server":{"undertow":{"io-threads":1, "worker-threads":2 }}}'
        - name: AMQP_USERNAME
          value: admin
        - name: AMQP_PASSWORD
          value: admin
```

8. Run the **mvn** command to deploy the quickstart to OpenShift server.

```
mvn fabric8:deploy -Popenshift
```

9. To verify that the quickstart is running successfully:
    a. Navigate to the OpenShift console.
    b. Select the project **amq-quickstart**.
    c. Click **Applications**.
    d. Select **Pods**.
    e. Click **fis-spring-boot-camel-am-1-xxxxx**.
    f. Click **Logs**.
    The output shows the messages are sent successfully.

```
10:17:59.825 [Camel (camel) thread #10 - timer://order] INFO  generate-order-route - Generating order order1379.xml
10:17:59.829 [Camel (camel) thread #8 - JmsConsumer[incomingOrders]] INFO  jms-cbr-route - Sending order order1379.xml to the UK
10:17:59.829 [Camel (camel) thread #8 - JmsConsumer[incomingOrders]] INFO  jms-cbr-route - Done processing order1379.xml
10:18:02.825 [Camel (camel) thread #10 - timer://order] INFO  generate-order-route - Generating order order1380.xml
10:18:02.829 [Camel (camel) thread #7 - JmsConsumer[incomingOrders]] INFO  jms-cbr-route - Sending order order1380.xml to another country
10:18:02.829 [Camel (camel) thread #7 - JmsConsumer[incomingOrders]] INFO jms-cbr-route - Done processing order1380.xml
```

10. To view the routes on the web interface, click **Open Java Console** and check the messages in the A-MQ queue.

CHAPTER 9. INTEGRATING SPRING BOOT WITH KUBERNETES

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

- Spring Boot Externalized Configuration
- Kubernetes ConfigMap
- Kubernetes Secrets

9.1. 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).

9.1.1. 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.

9.1.2. Kubernetes Secrets

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

9.1.3. 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.

9.1.4. Enabling Spring Boot with Kubernetes Integration

You can enable Kubernetes integration by adding it as a Maven dependency to **pom.xml** file.

Procedure

1. Enable the Kubernetes integration by adding the following Maven dependency to the pom.xml file of your Spring Boot Maven project.

```
<project ...>
```

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

2. To complete the integration,
   - Add some annotations to your Java source code
   - Create a Kubernetes ConfigMap object
   - Modify the OpenShift service account permissions to allow your application to read the ConfigMap object.

Additional resources

- For more details see Running Tutorial for ConfigMap Property Source.

9.2. RUNNING TUTORIAL FOR CONFIGMAP PROPERTY SOURCE

The following tutorial allows you to experiment with setting Kubernetes Secrets and ConfigMaps. Enable the Spring Cloud Kubernetes plug-in as explained in the Enabling Spring Boot with Kubernetes Integration to integrate Kubernetes configuration objects with Spring Boot Externalized Configuration.

9.2.1. Running Spring Boot Camel Config Quickstart

The following tutorial is based on the **spring-boot-camel-config-archetype** Maven archetype, which enables you to set up Kubernetes Secrets and ConfigMaps.

Procedure

1. Open a new shell prompt and enter the following Maven command to create a simple Camel Spring Boot project.

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
  -DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-catalog/2.2.0.fuse-740017-redhat-00003/archetypes-catalog-2.2.0.fuse-740017-redhat-00003-archetype-catalog.xml \
  -DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
  -DarchetypeArtifactId=spring-boot-camel-config-archetype \
  -DarchetypeVersion=2.2.0.fuse-740017-redhat-00003
```

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': : fuse74-configmap
```

```
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.example.fis: :
[INFO] Using property: spring-boot-version = 1.5.17.RELEASE
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse74-configmap
version: 1.0-SNAPSHOT
package: org.example.fis
spring-boot-version: 1.5.17.RELEASE
 Y: :
```

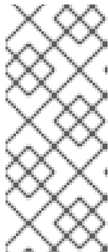
When prompted, enter **org.example.fis** for the **groupId** value and **fuse74-configmap** for the **artifactId** value. Accept the defaults for the remaining fields.

2. Log in to OpenShift and switch to the OpenShift project where you will deploy your application. For example, to log in as the **developer** user and deploy to the **test** project, enter the following commands:

```
oc login -u developer -p developer
oc project test
```

3. At the command line, change to the directory of the new **fuse74-configmap** project and create the Secret object for this application.

```
oc create -f sample-secret.yml
```



NOTE

It is necessary to create the Secret object *before* you deploy the application, otherwise the deployed container enters a wait state until the Secret becomes available. If you subsequently create the Secret, the container will come out of the wait state. For more information on how to set up Secret Object, see Setting up Secret.

4. Build and deploy the quickstart application. From the top level of the **fuse74-configmap** project, enter:

```
mvn fabric8:deploy -Popenshift
```

5. View the application log as follows.

   a. Open the OpenShift console in your browser and select the relevant project namespace (for example, **test**).

   b. Click in the center of the circular pod icon for the **fuse74-configmap** service.

   c. In the **Pods** view click on the pod **Name** to view the details of the running pod (alternatively, you will get straight through to the details page, if there is only one pod running).

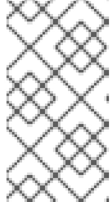
   d. Click the **Logs** tag to view the application log and scroll down to find the log messages generated by the Camel application.

6. The default recipient list, which is configured in **src/main/resources/application.properties**, sends the generated messages to two dummy endpoints: **direct:async-queue** and **direct:file**. This causes messages like the following to be written to the application log:

```
5:44:57.376 [Camel (camel) thread #0 - timer://order] INFO  generate-order-route - Generating message message-44, sending to the recipient list
15:44:57.378 [Camel (camel) thread #0 - timer://order] INFO  target-route-queue - ----> message-44 pushed to an async queue (simulation)
15:44:57.379 [Camel (camel) thread #0 - timer://order] INFO  target-route-queue - ----> Using username 'myuser' for the async queue
15:44:57.380 [Camel (camel) thread #0 - timer://order] INFO  target-route--file - ----> message-44 written to a file
```

7. Before you can update the configuration of the **fuse74-configmap** application using a ConfigMap object, you must give the **fuse74-configmap** application permission to view data from the OpenShift ApiServer. Enter the following command to give the **view** permission to the **fuse74-configmap** application's service account:

```
oc policy add-role-to-user view system:serviceaccount:test:qs-camel-config
```



NOTE

A service account is specified using the syntax **system:serviceaccount:PROJECT_NAME:SERVICE_ACCOUNT_NAME**. The **fis-config** deployment descriptor defines the **SERVICE_ACCOUNT_NAME** to be **qs-camel-config**.

8. To see the live reload feature in action, create a ConfigMap object as follows:

```
oc create -f sample-configmap.yml
```

The new ConfigMap overrides the recipient list of the Camel route in the running application, configuring it to send the generated messages to *three* dummy endpoints: **direct:async-queue**, **direct:file**, and **direct:mail**. For more information about ConfigMap object, see Setting up ConfigMap. This causes messages like the following to be written to the application log:

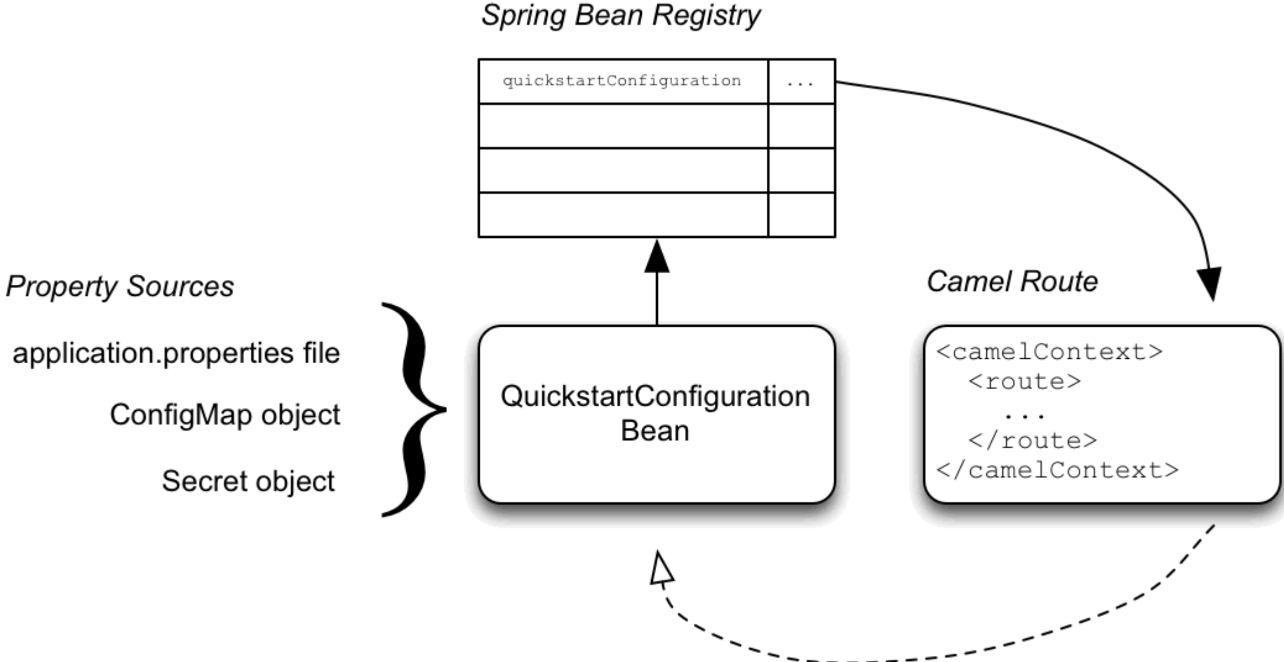
```
16:25:24.121 [Camel (camel) thread #0 - timer://order] INFO  generate-order-route - Generating message message-9, sending to the recipient list
16:25:24.124 [Camel (camel) thread #0 - timer://order] INFO  target-route-queue - ----> message-9 pushed to an async queue (simulation)
16:25:24.125 [Camel (camel) thread #0 - timer://order] INFO  target-route-queue - ----> Using username 'myuser' for the async queue
16:25:24.125 [Camel (camel) thread #0 - timer://order] INFO  target-route--file - ----> message-9 written to a file (simulation)
16:25:24.126 [Camel (camel) thread #0 - timer://order] INFO  target-route--mail - ----> message-9 sent via mail
```

9.2.2. Configuration Properties Bean

A configuration properties bean is a regular Java bean that can receive configuration settings by injection. It provides the basic interface between your Java code and the external configuration mechanisms.

Externalized Configuration and Bean Registry

Following image shows how Spring Boot Externalized Configuration works in the **spring-boot-camel-config** quickstart.



The configuration mechanism has the following main parts:

Property Sources

Provides property settings for injection into configuration. The default property source is the **application.properties** file for the application, and this can optionally be overridden by a ConfigMap object or a Secret object.

Configuration Properties bean

Receives configuraton updates from the property sources. A configuration properties bean is a Java bean decorated by the **@Configuration** and **@ConfigurationProperties** annotations.

Spring bean registry

With the requisite annotations, a configuration properties bean is registered in the Spring bean registry.

Integration with Camel bean registry

The Camel bean registry is automatically integrated with the Spring bean registry, so that registered Spring beans can be referenced in your Camel routes.

QuickstartConfiguration class

The configuration properties bean for the **fuse74-configmap** project is defined as the **QuickstartConfiguration** Java class (under the **src/main/java/org/example/fis/** directory), as follows:

```
package org.example.fis;

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

@Configuration (1)
@ConfigurationProperties(prefix = "quickstart") (2)
public class QuickstartConfiguration {
```

```
    /**
     * A comma-separated list of routes to use as recipients for messages.
     */
    private String recipients; (3)

    /**
     * The username to use when connecting to the async queue (simulation)
     */
    private String queueUsername; (4)

    /**
     * The password to use when connecting to the async queue (simulation)
     */
    private String queuePassword; (5)

    // Setters and Getters for Bean properties
    // NOT SHOWN
    ...
}
```

1. The **@Configuration** annotation causes the **QuickstartConfiguration** class to be instantiated and registered in Spring as the bean with ID, **quickstartConfiguration**. This automatically makes the bean accessible from Camel. For example, the **target-route-queue** route is able to access the **queueUserName** property using the Camel syntax **${bean:quickstartConfiguration?method=getQueueUsername}**.

2. The **@ConfigurationProperties** annotation defines a prefix, **quickstart**, that must be used when defining property values in a property source. For example, a properties file would reference the **recipients** property as **quickstart.recipients**.

3. The **recipient** property is injectable from property sources.

4. The **queueUsername** property is injectable from property sources.

5. The **queuePassword** property is injectable from property sources.

9.2.3. Setting up Secret

The Kubernetes Secret in this quickstart is set up in the standard way, apart from one additional required step: the Spring Cloud Kubernetes plug-in must be configured with the mount paths of the Secrets, so that it can read the Secrets at run time. To set up the Secret:

1. Create a Sample Secret Object
2. Configure volume mount for the Secret
3. Configure spring-cloud-kubernetes to read Secret properties

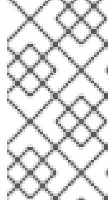
Sample Secret object

The quickstart project provides a sample Secret, **sample-secret.yml**, as follows. Property values in Secret objects are always base64 encoded (use the **base64** command-line utility). When the Secret is mounted in a pod's filesystem, the values are automatically decoded back into plain text.

sample-secret.yml file

```
apiVersion: v1
kind: Secret
metadata: (1)
  name: camel-config
type: Opaque
data:
  # The username is 'myuser'
  quickstart.queue-username: bXl1c2VyCg== (2)
  quickstart.queue-password: MWYyZDFlMmU2N2Rm (3)
```

1. metadata.name: Identifies the Secret. Other parts of the OpenShift system use this identifier to reference the Secret.
2. quickstart.queue-username: Is meant to be injected into the **queueUsername** property of the **quickstartConfiguration** bean. The value *must* be base64 encoded.
3. quickstart.queue-password: Is meant to be injected into the **queuePassword** property of the **quickstartConfiguration** bean. The value *must* be base64 encoded.



NOTE

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

Configure volume mount for the Secret

The application must be configured to load the Secret at run time, by configuring the Secret as a volume mount. After the application starts, the Secret properties then become available at the specified location in the filesystem. The **deployment.yml** file for the application is located under **src/main/fabric8/** directory, which defines the volume mount for the Secret.

deployment.yml file

```
spec:
  template:
    spec:
      serviceAccountName: "qs-camel-config"
      volumes: (1)
        - name: "camel-config"
          secret:
            # The secret must be created before deploying this application
            secretName: "camel-config"
      containers:
        -
          volumeMounts: (2)
            - name: "camel-config"
              readOnly: true
              # Mount the secret where spring-cloud-kubernetes is configured to read it
              # see src/main/resources/bootstrap.yml
```

```
            mountPath: "/etc/secrets/camel-config"
          resources:
#           requests:
#             cpu: "0.2"
#             memory: 256Mi
#           limits:
#             cpu: "1.0"
#             memory: 256Mi
          env:
          - name: SPRING_APPLICATION_JSON
            value: '{"server":{"undertow":{"io-threads":1, "worker-threads":2 }}}'
```

1. In the **volumes** section, the deployment declares a new volume named **camel-config**, which references the Secret named **camel-config**.
2. In the **volumeMounts** section, the deployment declares a new volume mount, which references the **camel-config** volume and specifies that the Secret volume should be mounted to the path **/etc/secrets/camel-config** in the pod's filesystem.

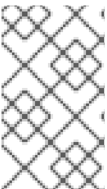
Configuring spring-cloud-kubernetes to read Secret properties

To integrate secrets with Spring Boot externalized configuration, the Spring Cloud Kubernetes plug-in must be configured with the secret's mount path. Spring Cloud Kubernetes reads the secrets from the specified location and makes them available to Spring Boot as property sources. The Spring Cloud Kubernetes plug-in is configured by settings in the **bootstrap.yml** file, located under **src/main/resources** in the quickstart project.

bootstrap.yml file

```
# Startup configuration of Spring-cloud-kubernetes
spring:
  application:
    name: camel-config
  cloud:
    kubernetes:
      reload:
        # Enable live reload on ConfigMap change (disabled for Secrets by default)
        enabled: true
      secrets:
        paths: /etc/secrets/camel-config
```

The **spring.cloud.kubernetes.secrets.paths** property specifies the list of paths of secrets volume mounts in the pod.



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.

9.2.4. Setting up ConfigMap

In addition to creating a ConfigMap object and setting the view permission appropriately, the

integration with Spring Cloud Kubernetes requires you to match the ConfigMap's **metadata.name** with the value of the **spring.application.name** property configured in the project's **bootstrap.yml** file. To set up the ConfigMap:

- Create Sample ConfigMap Object
- Set up the view permission
- Configure the Spring Cloud Kubernetes plug-in

Sample ConfigMap object

The quickstart project provides a sample ConfigMap, **sample-configmap.yml**.

```
kind: ConfigMap
apiVersion: v1
metadata: (1)
  # Must match the 'spring.application.name' property of the application
  name: camel-config
data:
  application.properties: | (2)
    # Override the configuration properties here
    quickstart.recipients=direct:async-queue,direct:file,direct:mail (3)
```

1. metadata.name: Identifies the ConfigMap. Other parts of the OpenShift system use this identifier to reference the ConfigMap.
2. data.application.properties: This section lists property settings that can override settings from the original **application.properties** file that was deployed with the application.
3. quickstart.recipients: Is meant to be injected into the **recipients** property of the **quickstartConfiguration** bean.

Setting the view permission

As shown in the deployment.yml file for the Secret, the **serviceAccountName** is set to **qs-camel-config** in the project's **deployment.yml** file. Hence, you need to enter the following command to enable the **view** permission on the quickstart application (assuming that it deploys into the **test** project namespace):

```
oc policy add-role-to-user view system:serviceaccount:test:qs-camel-config
```

Configuring the Spring Cloud Kubernetes plug-in

The Spring Cloud Kubernetes plug-in is configured by the following settings in the **bootstrap.yml** file.

spring.application.name

This value must match the **metadata.name** of the ConfigMap object (for example, as defined in **sample-configmap.yml** in the quickstart project). It defaults to **application**.

spring.cloud.kubernetes.reload.enabled

Setting this to **true** enables dynamic reloading of ConfigMap objects.

For more details about the supported properties, see PropertySource Reload Configuration Properties.

9.3. USING 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:

- Apply Individual Properties
- Apply Property Named application.yaml
- Apply Property Named application.properties

9.3.1. Applying 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.max**

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
```

9.3.2. Applying 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
```

9.3.3. Applying 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
```

9.3.4. Deploying a ConfigMap

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

Procedure

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)
   ```

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

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**.

9.4.1. 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:
  pg.username: dXNlcgo=
  pg.password: cGdhZG1pbgo=
```

9.4.2. Consuming the Secrets

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

- 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
  ```

- By setting a named secret:

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

- By defining a list of labels:

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

9.4.3. Configuration Properties for Secrets PropertySource

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

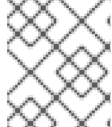
Sets the labels used to lookup secrets. This property behaves as defined by Map-based binding. Type is **java.util.Map** and default is **null**.

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.

9.5. USING 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.

9.5.1. Enabling PropertySource Reload

The **PropertySource reload** feature of Spring Cloud Kubernetes is disabled by default.

Procedure

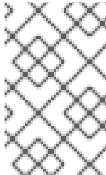
1. Navigate to **src/main/resources** directory of the quickstart project and open the **bootstrap.yml** file.
2. Change the configuration property **spring.cloud.kubernetes.reload.enabled=true**.

9.5.2. Levels of PropertySource Reload

The following levels of reload are supported for 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.

9.5.3. Example of PropertySource Reload

The following example explains what happens when the reload feature is enabled.

Procedure

1. Assume 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

}
```

2. To see the changes that are happening, create another bean that prints the message periodically as shown below.

```
@Component
public class MyBean {

    @Autowired
    private MyConfig config;

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

3. You can change the message printed by the application by using a ConfigMap as shown below.

```
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.

9.5.4. PropertySource Reload Operating Modes

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.

9.5.5. PropertySource Reload Configuration Properties

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 10. DEVELOPING AN APPLICATION FOR THE KARAF IMAGE

This tutorial shows how to create and deploy an application for the Karaf image.

10.1. CREATING A KARAF PROJECT USING MAVEN ARCHETYPE

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

Procedure

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

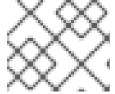
```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
  -
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-catalog/2.2.0.fuse-740017-redhat-00003/archetypes-catalog-2.2.0.fuse-740017-redhat-00003-archetype-catalog.xml \
  -DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
  -DarchetypeArtifactId=karaf-camel-log-archetype \
  -DarchetypeVersion=2.2.0.fuse-740017-redhat-00003
```

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': : fuse74-karaf-camel-log
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.example.fis: :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse74-karaf-camel-log
version: 1.0-SNAPSHOT
package: org.example.fis
 Y: :
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse74-karaf-camel-log** for the **artifactId** value. Accept the defaults for the remaining fields.

4. Follow the instructions in the quickstart on how to build and deploy the example.



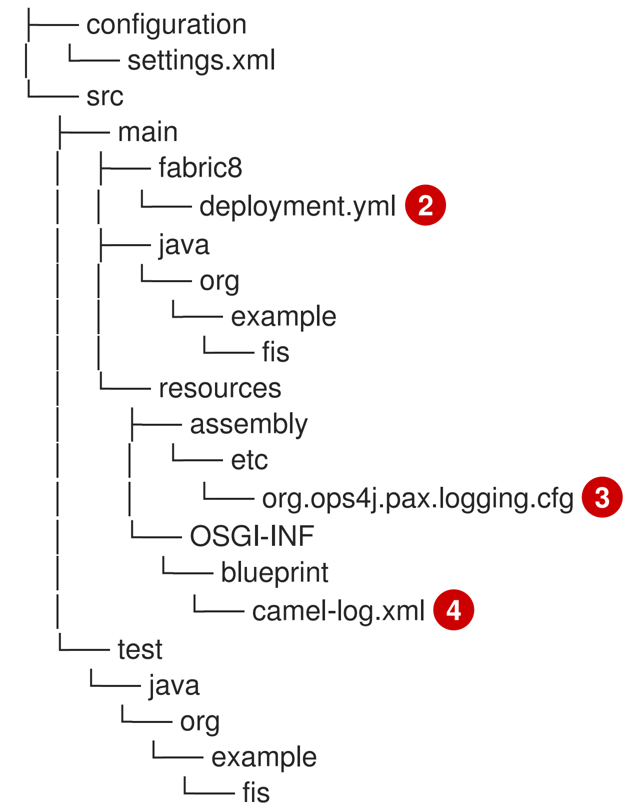
NOTE

For the full list of available Karaf archetypes, see Karaf Archetype Catalog.

10.2. STRUCTURE OF THE CAMEL KARAF APPLICATION

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

```
├── pom.xml 1
├── README.md
```



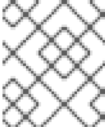
```
├── configuration
│   └── settings.xml
└── src
    ├── main
    │   ├── fabric8
    │   │   └── deployment.yml 2
    │   ├── java
    │   │   └── org
    │   │       └── example
    │   │           └── fis
    │   └── resources
    │       ├── assembly
    │       │   └── etc
    │       │       └── org.ops4j.pax.logging.cfg 3
    │       └── OSGI-INF
    │           └── blueprint
    │               └── camel-log.xml 4
    └── test
        └── java
            └── org
                └── example
                    └── fis
```

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

1. 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>
```

2. 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.

3. org.ops4j.pax.logging.cfg: Demonstrates how to customize log levels, sets logging level to DEBUG instead of the default INFO.

4. camel-log.xml: Contains the source code of the application.

10.3. KARAF ARCHETYPE CATALOG

The Karaf archetype catalog includes the following examples.

Table 10.1. Karaf Maven Archetypes

Name	Description
karaf-camel-amq-archetype	Demonstrates how to send and receive messages to an Apache ActiveMQ message broker, using the Camel amq component.
karaf-camel-log-archetype	Demonstrates a simple Apache Camel application that logs a message to the server log every 5th second.
karaf-camel-rest-sql-archetype	Demonstrates how to use SQL via JDBC along with Camel's REST DSL to expose a RESTful API.
karaf-cxf-rest-archetype	Demonstrates how to create a RESTful(JAX-RS) web service using CXF and expose it through the OSGi HTTP Service.

10.4. USING 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.

10.4.1. Adding Fabric8 Karaf Features

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

Procedure

1. Open your project's **pom.xml** file and add **fabric8-karaf-features** dependency.

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

The fabric8 karaf features will be installed into the Karaf server.

10.4.2. Adding Fabric8 Karaf Core Bundle Functionality

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

Procedure

1. Open your project's **pom.xml** and add **fabric8-karaf-core** to **startupFeatures** section.

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

   This will add the **fabric8-karaf-core** feature in a custom Karaf distribution.

10.4.3. Setting the Property Placeholder Service Options

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

Procedure

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

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

2. Mount the 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
   ```

10.4.4. Adding a Custom Property Placeholder Resolver

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.

Procedure

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]
```

10.4.5. List of Resolution Strategies

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**	look up the property from OS environment variables.
`sys	**sys:java.version**	look up the property from Java JVM system properties.
`service	**service:amq**	look up the property from OS environment variables using the service naming convention.
service.host	**service.host:amq**	look up the property from OS environment variables using the service naming convention returning the hostname part only.
service.port	**service.port:amq**	look up the property from OS environment variables using the service naming convention returning the port part only.
k8s:map	**k8s:map:myMap/myKey**	look up the property from a Kubernetes ConfigMap (via API)
k8s:secret	**k8s:secret:amq/password**	look up the property from a Kubernetes Secrets (via API or volume mounts)

10.4.6. List of Property Placeholder Service Options

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 where secrets are mapped
fabric8.k8s.secrets.api.enabled	false	Enable/Disable consuming secrets via APIs

10.5. ADDING FABRIC8 KARAF CONFIG ADMIN SUPPORT

10.5.1. Adding Fabric8 Karaf Config Admin Support

You can add Fabric8 Karaf Config Admin support to your custom Karaf distribution.

Procedure

- Open your project's **pom.xml** and add **fabric8-karaf-cm** to **startupFeatures** section.

  pom.xml

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

10.5.2. Adding ConfigMap Injection

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

Procedure

1. To be added by the ConfigAdmin bridge, a ConfigMap has to be labeled with **karaf.pid**. The **karaf.pid** value corresponds to the pid of your component. For example,

```
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
```

2. To define your configuration, you can use single property names. Individual properties work for most cases. 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
```

10.5.3. 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
```

10.5.4. Config Property Placeholders

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

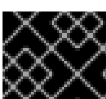
```
    <?xml version="1.0" encoding="UTF-8"?>

    <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>
```

10.5.5. Fabric8 Karaf Config Admin options

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

Name	Default	Description
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 (that is, a ConfigMap that needs to be selected must have that label; the value of which determines to what PID it gets associated)
fabric8.pid.filters	empty	Define additional conditions for the ConfigAdmin bridge to select a ConfigMap. The supported syntax is: • Conditions on different labels are separated by "," and are intended in AND between each other. • Inside a label, separated by ";" there might be conditions on the label value which are considered in OR with each other. For example, a filter like **-Dfabric8.pid.filters=appName=A;B,database.name=my.oracle.datasource** translates to "give me all the ConfigMaps that have a label appName with values A or B and a label database.name equals to my.oracle.datasource".



IMPORTANT

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

10.6. ADDING 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.

Procedure

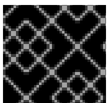
- To include the feature for blueprint support in your custom Karaf distribution, add **fabric8-karaf-blueprint** to **startupFeatures** section in your project **pom.xml**.

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

Example

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>
```

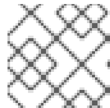


IMPORTANT

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

10.7. ENABLING 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.

Procedure

1. Open you project's **pom.xml** and add **fabric8-karaf-checks** feature in the **startupFeatures** section.

   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.

10.8. ADDING CUSTOM HEALTH CHECKS

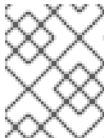
You can provide additional custom heath checks to prevent the Karaf server from receiving user traffic before it is ready to process the requests. To enable custom health checks you need to implement the **io.fabric8.karaf.checks.HealthChecker** or **io.fabric8.karaf.checks.ReadinessChecker** interfaces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

Procedure

- 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.

Example

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 11. DEVELOPING AN APPLICATION FOR THE JBOSS EAP IMAGE

To develop Fuse applications on JBoss EAP, an alternative is to use the S2I source workflow to create an OpenShift project for Red Hat Camel CDI with EAP.

Prerequisites

- Ensure that OpenShift is running correctly and the Fuse image streams are already installed in OpenShift. See Getting Started for Administrators.
- Ensure that Maven Repositories are configured for fuse, see Configuring Maven Repositories.

11.1. CREATING A JBOSS EAP PROJECT USING THE S2I SOURCE WORKFLOW

To develop Fuse applications on JBoss EAP, an alternative is to use the S2I source workflow to create an OpenShift project for Red Hat Camel CDI with EAP.

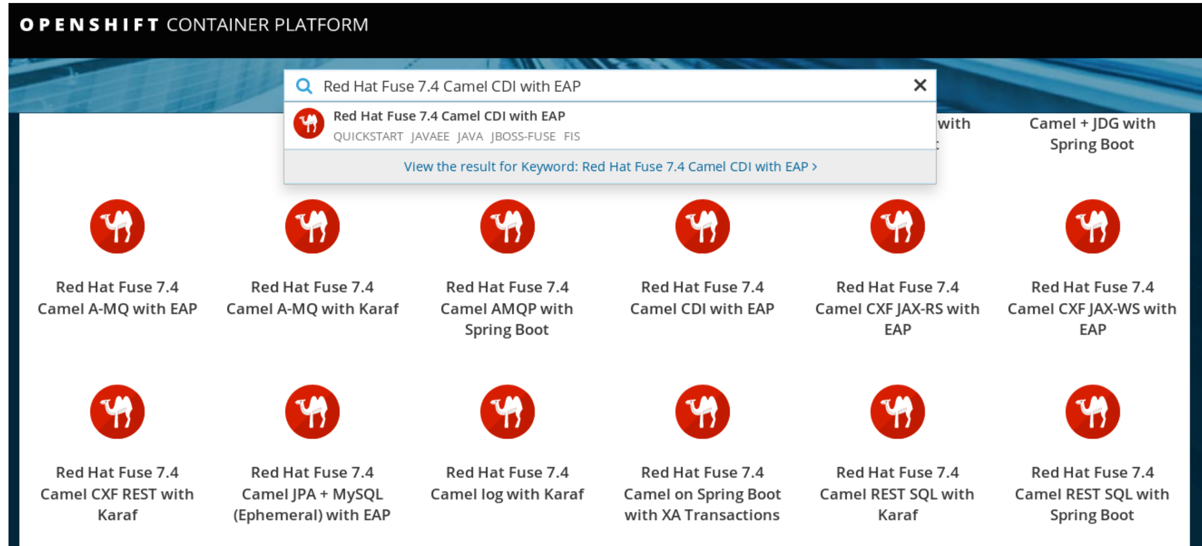
Procedure

1. Add the **view** role to the default service account to enable clustering. This grants the user the **view** access to the **default** service account. Service accounts are required in each project to run builds, deployments, and other pods. Enter the following **oc** client commands in a shell prompt:

```
oc login -u developer -p developer
oc policy add-role-to-user view -z default
```

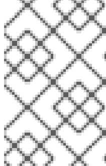
2. Navigate to the OpenShift console in your browser (https://OPENSHIFT_IP_ADDR:8443, replace **OPENSHIFT_IP_ADDR** with the IP address that was displayed in the case of CDK) and log in to the console with your credentials (for example, with username **developer** and password, **developer**).

3. In the Catalog search field, enter **Red Hat Fuse 7.4 Camel CDI with EAP** as the search string and select the **Red Hat Fuse 7.4 Camel CDI with EAP**template.



4. The **Information** step of the template wizard opens. Click **Next**.

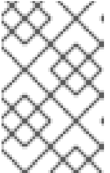
5. The **Configuration** step of the template wizard opens. From the **Add to Project** dropdown, select **My Project**.



NOTE

Alternatively, if you prefer to create a new project for this example, select **Create Project** from the **Add to Project** dropdown. A **Project Name** field then appears for you to fill in the name of the new project.

6. You can accept the default values for the rest of the settings in the **Configuration** step. Click **Create**.



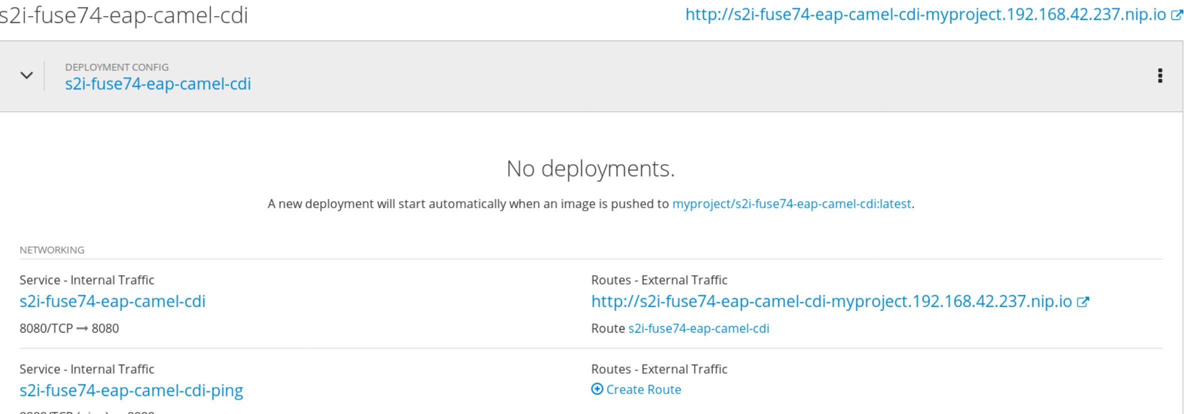
NOTE

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.

7. The **Results** step of the template wizard opens. Click **Close**.

8. In the right-hand **My Projects** pane, click **My Project**. The **Overview** tab of the **My Project** project opens, showing the **s2i-fuse74-eap-camel-cdi** application.

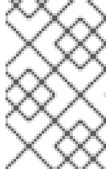
9. Click the arrow on the left of the **s2i-fuse74-eap-camel-cdi** deployment to expand and view the details of this deployment, as shown.



10. In this view, you can see the build log. If the build should fail for any reason, the build log can help you to diagnose the problem.

Build #2 is running ... created 4 minutes ago View Full Log

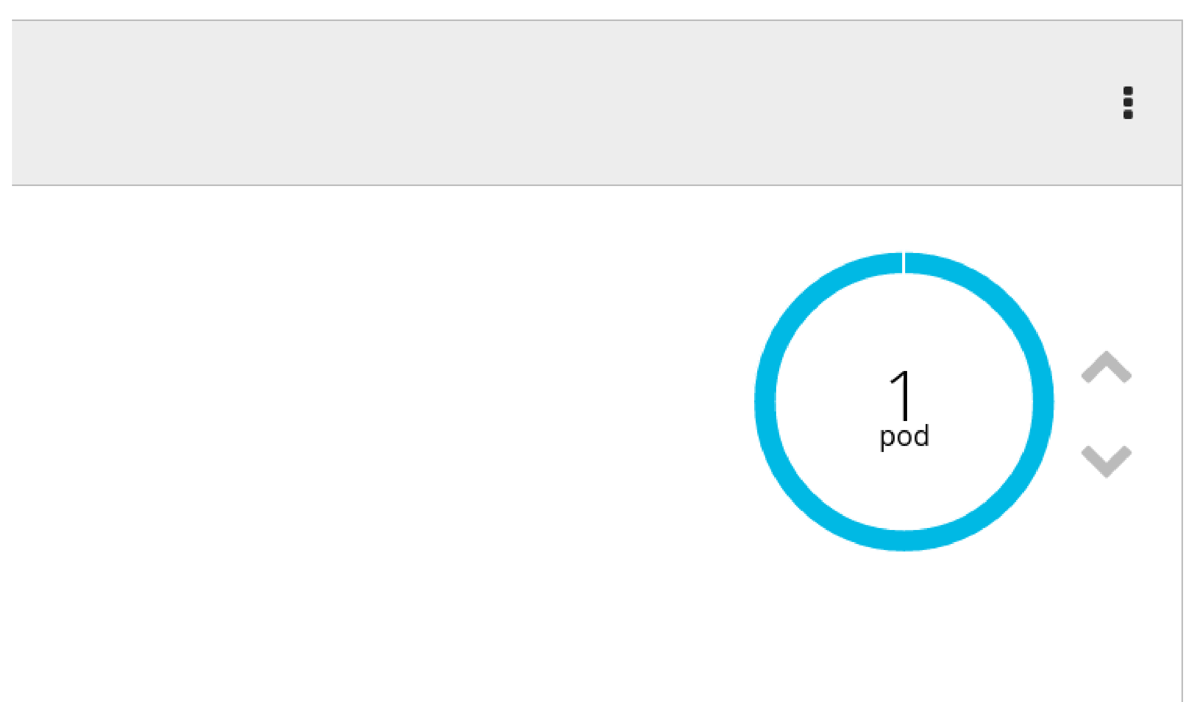
```
[INFO] Downloading: https://repo1.maven.org/maven2/org/codehaus/plexus…
[INFO] Downloaded: https://repo1.maven.org/maven2/org/codehaus/plexus/…
[INFO] Downloading: https://repo1.maven.org/maven2/org/sonatype/spice/…
[INFO] Downloaded: https://repo1.maven.org/maven2/org/sonatype/spice/s…
[INFO] Downloading: https://repo1.maven.org/maven2/org/sonatype/forge/…
[INFO] Downloaded: https://repo1.maven.org/maven2/org/sonatype/forge/f…
[INFO] Downloading: https://repo1.maven.org/maven2/org/codehaus/plexus…
```



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.

11. If the build completes successfully, the pod icon shows as a blue circle with **1 pod** running.



12. To open the application, click the link that is shown above the application details, which has the form **http://s2i-fuse74-eap-camel-cdi-myproject.IP_ADDRESS.nip.io/**. This shows a message like the following in your browser:

```
Hello world from 172.17.0.3
```

You can also specify a name using the **name** parameter in the URL. For example, if you enter the URL, **http://s2i-fuse74-eap-camel-cdi-myproject.IP_ADDRESS.nip.io/?name=jdoe**, in your browser you see the response:

```
Hello jdoe from 172.17.0.3
```

13. Click **Overview** on the left-hand navigation bar to return to the overview of the applications in the **My Project** 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-fuse74-eap-camel-cdi-1?**, click **Scale Down**.

14. *(Optional)* If you are using CDK, you can shut down the virtual OpenShift Server completely by returning to the shell prompt and entering the following command:

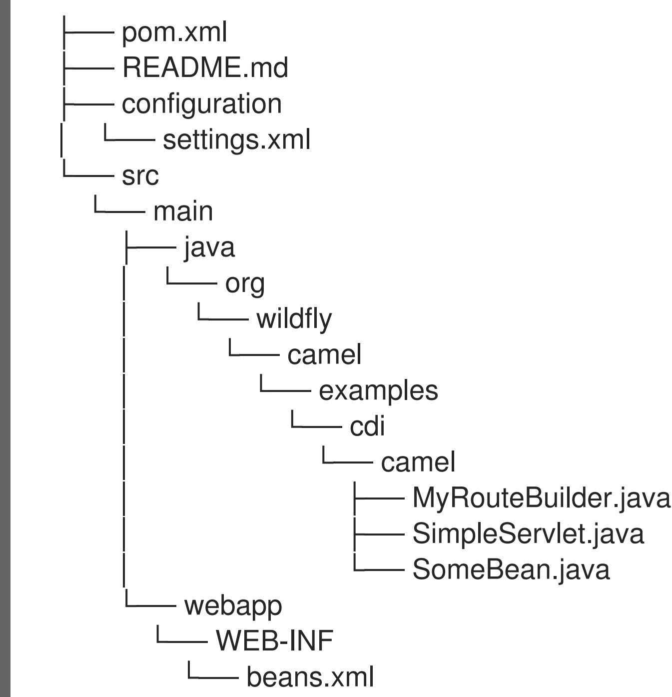
```
minishift stop
```

11.2. STRUCTURE OF THE JBOSS EAP APPLICATION

You can find the source code for the Red Hat Fuse 7.4 Camel CDI with EAP example at the following location:

> https://github.com/wildfly-extras/wildfly-camel-examples/tree/wildfly-camel-examples-5.2.0.fuse-720021/camel-cdi

The directory structure of the Camel on EAP application is as follows:



```
├── pom.xml
├── README.md
├── configuration
│   └── settings.xml
└── src
    └── main
        ├── java
        │   └── org
        │       └── wildfly
        │           └── camel
        │               └── examples
        │                   └── cdi
        │                       └── camel
        │                           ├── MyRouteBuilder.java
        │                           ├── SimpleServlet.java
        │                           └── SomeBean.java
        └── webapp
            └── WEB-INF
                └── beans.xml
```

Where the following files are important for developing a JBoss EAP application:

pom.xml

Includes additional dependencies.

11.3. JBOSS EAP QUICKSTART TEMPLATES

The following S2I templates are provided for Fuse on JBoss EAP:

Table 11.1. JBoss EAP S2I Templates

Name	Description
JBoss Fuse 7.4 Camel A-MQ with EAP (**eap-camel-amq-template**)	Demonstrates using the camel-activemq component to connect to an AMQ message broker running in OpenShift. It is assumed that the broker is already deployed.
Red Hat Fuse 7.4 Camel CDI with EAP (**eap-camel-cdi-template**)	Demonstrates using the camel-cdi component to integrate CDI beans with Camel routes.
Red Hat Fuse 7.4 CXF JAX-RS with EAP (**eap-camel-cxf-jaxrs-template**)	Demonstrates using the camel-cxf component to produce and consume JAX-RS REST services.

Name	Description
Red Hat Fuse 7.4 CXF JAX-WS with EAP (**eap-camel-cxf-jaxws-template**)	Demonstrates using the camel-cxf component to produce and consume JAX-WS web services.
Red Hat Fuse 7.4 Camel JPA + MySQL (Ephemeral) with EAP **eap-camel-jpa-template**	Demonstrates how to connect a Camel application with Red Hat Fuse on EAP to a MySQL database and expose a REST API. This example creates two containers, one container to run as a MySQL server, and another running the Camel application which acts as as a client to the database.

CHAPTER 12. USING PERSISTENT STORAGE IN FUSE ON OPENSHIFT

Fuse on OpenShift 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-formatted 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.

12.1. ABOUT VOLUMES AND VOLUME TYPES

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.

12.2. ABOUT 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 Configuring Persistent Storage for creating various types of PersistentVolumes.

12.3. CONFIGURING PERSISTENT VOLUME

You can provision a Persistent Volume by creating a configuration file. This storage then can be accessed by creating a PersistentVolume Claim.

Procedure

1. Create a configuration file named **pv.yaml** using the sample configuration below. This 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.

2. Create the **PersistentVolume**.

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

3. Verify the creation of **PersistentVolume**. This will list all the **PersistentVolumes** configured in your OpenShift cluster:

```
oc get pv
```

12.4. CREATING 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.

Procedure

- 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
```

12.5. USING PERSISTENT VOLUMES IN PODS

Pods use Volume Mounts to define the filesystem mount location and Volumes to define reference **PersistentVolumeClaims**.

Procedure

1. Create a sample container configuration as shown below which 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.

2. Add this configuration in the file **src/main/fabric8/deployment.yml** in a Fuse on OpenShift application and create OpenShift resources using command:

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

3. Verify that the created **DeploymentConfiguration** has the volume mount and the volume.

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

   For Fuse on OpenShift quickstarts, replace the **<application-dc-name>** with the Maven project name, for example **spring-boot-camel**.

CHAPTER 13. PATCHING FUSE ON OPENSHIFT

You might need to perform one or more of the following tasks to bring the Fuse on OpenShift product up to the latest patch level:

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

Patch Application Dependencies using BOM
: Update the dependencies in your project POM file, so that your application uses patched versions of the Maven artifacts.

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

13.1. IMPORTANT NOTE ON BOMS AND MAVEN DEPENDENCIES

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

It is important to upgrade all of the Maven dependencies for Fuse on OpenShift together, so that your project uses dependencies that are all from the same patch version. The Fuse on OpenShift 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* Fuse on OpenShift 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 Fuse on OpenShift. When you update the version of a BOM file, you automatically update the versions for all the Fuse on OpenShift Maven artifacts in your project's POM.

The POM file that is generated by a Fuse on OpenShift Maven archetype or by a Fuse on OpenShift template has a standard layout that uses a BOM file and defines the versions of certain required plugins. 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.

13.2. PATCHING THE FUSE ON OPENSHIFT IMAGES

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

- **fuse7-java-openshift**
- **fuse7-karaf-openshift**
- **fuse7-eap-openshift**
- **fuse7-console**
- **apicurito-ui**

- **fuse-apicurito-generator**

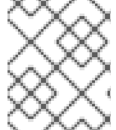
Procedure

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

```
$ oc get is -n openshift
NAME                    DOCKER REPO                                      TAGS
UPDATED
fuse7-console            172.30.1.1:5000/openshift/fuse7-console
1.0,1.1,1.2,1.3,1.4        About an hour ago
fuse7-eap-openshift        172.30.1.1:5000/openshift/fuse7-eap-openshift
1.0,1.1,1.2,,1.3,1.4       About an hour ago
fuse7-java-openshift       172.30.1.1:5000/openshift/fuse7-java-openshift
1.0,1.1,1.2,1.3,1.4        About an hour ago
fuse7-karaf-openshift      172.30.1.1:5000/openshift/fuse7-karaf-openshift
1.0,1.1,1.2,1.3,1.4        About an hour ago...
fuse-apicurito-generator     172.30.1.1:5000/openshift/fuse-apicurito-generator
1.2,1.3,1.4              About an hour ago...
apicurito-ui             172.30.1.1:5000/openshift/apicurito-ui             1.2,1.3,1.4
About an hour ago...
```

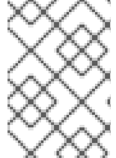
2. You can now update each image stream one at a time:

```
oc import-image -n openshift fuse7-java-openshift:1.4
oc import-image -n openshift fuse7-karaf-openshift:1.4
oc import-image -n openshift fuse7-eap-openshift:1.4
oc import-image -n openshift fuse7-console:1.4
oc import-image -n openshift apicurito-ui:1.4
oc import-image -n openshift fuse-apicurito-generator:1.4
```



NOTE

The version tags in the image stream have the form **1.4-<BUILDNUMBER>**. When you specify the tag as **1.4**, you will get the latest build in the **1.4** stream.



NOTE

You can also configure your Fuse applications so that a rebuild is automatically triggered whenever a new Fuse on OpenShift image becomes available. For details, see the section Setting Deployment Triggers in the OpenShift Container Platform 3.11 Developer Guide_.

13.3. PATCHING THE FUSE ON OPENSHIFT TEMPLATES

You must update the Fuse on OpenShift templates to the latest patch level, to ensure that new template-based projects are built using the correct patched dependencies.

Procedure

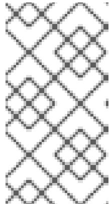
1. You need administrator privileges to update the Fuse on OpenShift templates. Log in to the OpenShift Server as an administrator, as follows:

```
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 Fuse on OpenShift templates. Enter the following commands at a command prompt:

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-740025-redhat-00003
oc replace --force -n openshift -f ${BASEURL}/quickstarts/eap-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/eap-camel-cdi-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/eap-camel-cxf-jaxrs-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/eap-camel-cxf-jaxws-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/eap-camel-jpa-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf-camel-log-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf-camel-rest-sql-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf-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-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.

13.4. PATCH APPLICATION DEPENDENCIES USING BOM

If your application **pom.xml** file is configured to use the new-style BOM, follow the instructions in this section to upgrade the Maven dependencies.

13.4.1. Updating Dependencies in a Spring Boot Application

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

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

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

  <fuse.version>7.4.0.fuse-740036-redhat-00002</fuse.version>
  ...
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.redhat-fuse</groupId>
      <artifactId>fuse-springboot-bom</artifactId>
      <version>${fuse.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
...
<build>
  ...
  <plugins>
    <!-- Core plugins -->
    ...
    <plugin>
      <groupId>org.jboss.redhat-fuse</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      ...
      <version>${fuse.version}</version>
    </plugin>
  </plugins>
</build>

<profiles>
  <profile>
    <id>openshift</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.jboss.redhat-fuse</groupId>
          <artifactId>fabric8-maven-plugin</artifactId>
          ...
          <version>${fuse.version}</version>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
</project>
```

When it comes to patching or upgrading the application, the following version settings are important:

fuse.version

Defines the version of the new-style **fuse-springboot-bom** BOM, as well as the versions of the **fabric8-maven-plugin** plugin and the **spring-boot-maven-plugin** plugin.

13.4.2. Updating Dependencies in a Karaf Application

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

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

    <fuse.version>7.4.0.fuse-740036-redhat-00002</fuse.version>
    ...
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-karaf-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>${fuse.version}</version>
        ...
      </plugin>
      ...
      <plugin>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fabric8-maven-plugin</artifactId>
        <version>${fuse.version}</version>
        ...
      </plugin>
    </plugins>
  </build>

</project>
```

When it comes to patching or upgrading the application, the following version settings are important:

fuse.version

Defines the version of the new-style **fuse-karaf-bom** BOM, as well as the versions of the **fabric8-maven-plugin** plugin and the **karaf-maven-plugin** plugin.

13.4.3. Updating Dependencies in a JBoss EAP Application

The following code fragment shows the standard layout of a POM file for a JBoss EAP application in Fuse on OpenShift, highlighting some important property settings:

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

    <fuse.version>7.4.0.fuse-740036-redhat-00002</fuse.version>
    ...
  </properties>

  <!-- Dependency Management -->
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-eap-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

When it comes to patching or upgrading the application, the following version settings are important:

fuse.version

Defines the version of the **fuse-eap-bom** BOM file (which replaces the old-style **wildfly-camel-bom** BOM file). By updating the BOM version to a particular patch version, you are effectively updating all of the Fuse on JBoss EAP Maven dependencies as well.

13.5. AVAILABLE BOM VERSIONS

The following table shows the new-style BOM versions corresponding to different patch releases of Red Hat Fuse.

Table 13.1. Red Hat Fuse Releases and Corresponding New-Style BOM Version

Red Hat Fuse Release	**org.jboss.redhat-fuse** BOM Version
Red Hat Fuse 7.0.0 GA	7.4.0.fuse-740036-redhat-00002
Red Hat Fuse 7.0.1 patch	7.0.1.fuse-000008-redhat-4

To upgrade your application POM to a specific Red Hat Fuse patch release, set the **fuse.version** property to the corresponding BOM version.

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

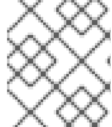
The Spring Boot Plugin includes the following goals:

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**. The following example illustrates the usage of the **spring-boot-maven-plugin** for Spring Boot.

- Spring Boot 2 Example
- Spring Boot 1 Example



NOTE

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

A.3.1. Spring Boot Maven Plugin for Spring Boot 2

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

  <groupId>com.redhat.fuse</groupId>
  <artifactId>spring-boot-camel</artifactId>
  <version>1.0-SNAPSHOT</version>

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

        <!-- configure the Fuse version you want to use here -->
        <fuse.bom.version>7.4.0.fuse-sb2-740019-redhat-00005</fuse.bom.version>

        <!-- maven plugin versions -->
        <maven-compiler-plugin.version>3.7.0</maven-compiler-plugin.version>
```

```
    <maven-surefire-plugin.version>2.19.1</maven-surefire-plugin.version>
  </properties>


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

    <plugins>
      <plugin>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>${fuse.bom.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

  <repositories>
    <repository>
      <id>redhat-ga-repository</id>
      <url>https://maven.repository.redhat.com/ga</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
    <repository>
      <id>redhat-ea-repository</id>
      <url>https://maven.repository.redhat.com/earlyaccess/all</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>
      <id>redhat-ga-repository</id>
      <url>https://maven.repository.redhat.com/ga</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
```

```
        </pluginRepository>
        <pluginRepository>
            <id>redhat-ea-repository</id>
            <url>https://maven.repository.redhat.com/earlyaccess/all</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
    </pluginRepositories>
</project>
```

A.3.2. Spring Boot Maven Plugin for Spring Boot 1

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

  <groupId>com.redhat.fuse</groupId>
  <artifactId>spring-boot-camel</artifactId>
  <version>1.0-SNAPSHOT</version>

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

        <!-- configure the Fuse version you want to use here -->
        <fuse.bom.version>7.4.0.fuse-740036-redhat-00002</fuse.bom.version>

        <!-- maven plugin versions -->
        <maven-compiler-plugin.version>3.7.0</maven-compiler-plugin.version>
        <maven-surefire-plugin.version>2.19.1</maven-surefire-plugin.version>
    </properties>


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

        <plugins>
            <plugin>
                <groupId>org.jboss.redhat-fuse</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>${fuse.bom.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

```
  <repositories>
    <repository>
      <id>redhat-ga-repository</id>
      <url>https://maven.repository.redhat.com/ga</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
    <repository>
      <id>redhat-ea-repository</id>
      <url>https://maven.repository.redhat.com/earlyaccess/all</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>
      <id>redhat-ga-repository</id>
      <url>https://maven.repository.redhat.com/ga</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
    <pluginRepository>
      <id>redhat-ea-repository</id>
      <url>https://maven.repository.redhat.com/earlyaccess/all</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</project>
```

APPENDIX B. USING 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. The finished assembly contains all of the essential components of a Karaf installation (including the contents of the etc/, data/, lib, and system directories), but stripped down to the bare minimum required to run your application.

B.1. MAVEN DEPENDENCIES

Maven dependencies in a **karaf-assembly** project are either feature repositories (classifier **features**) or kar archives.

- Feature repositories are installed in the maven structured system/internal repository.
- Kar archives have their content unpacked on top of the server as well as have the contained feature repositories installed.

B.1.1. Maven dependency scopes

The Maven scope of a dependency determines if its feature repository is listed in the features service configuration file **etc/org.apache.karaf.features.cfg** (under the featuresRepositories property). These scopes are:

- compile (default): All the features in the repository (or for a kar repositories) will be installed into the **startup.properties**. The feature repository is not listed in the features service configuration file.
- runtime: As boot stage in **karaf-maven-plugin**.
- Provided: As install stage in **karaf-maven-plugin**.

B.2. KARAF MAVEN PLUGIN CONFIGURATION

The **karaf-maven-plugin** defines three stages related with Maven scopes. The plugin configuration controls how features are installed using these elements by referring to features from installed feature repositories:

- Startup stage: **etc/startup.properties**
  In this stage, startup features, startup profiles, and startup bundles are used to prepare a list of bundles to be included in **etc/startup.properties**. This will result in the feature bundles being listed in **etc/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, for example, **<startupFeature>foo</startupFeature>**.
- Boot stage: **etc/org.apache.karaf.features.cfg**
  This stage manages features available in **featuresBoot** property and repositories in **featuresRepositories** property. This will result in the feature name added to boot-features in the features service configuration file and all the bundles in the feature copied into the **system** internal repository. You can use **feature_name** or **feature_name/feature_version** formats, for example, **<bootFeature>bar</bootFeature>**.
- Install stage:
  This stage installs the artifacts in **${karaf.home}/${karaf.default.repository}**. This will result in all the bundles in the feature being installed in the **system** internal repository. Therefore at runtime the feature may be installed without access to external repositories. You can use

feature_name or **feature_name/feature_version** formats, for example, **<installedFeature>baz</installedFeature>**.

- Libraries
  The plugin accepts the libraries element, which can have one or more library child elements that specify a library URL.

  Example

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

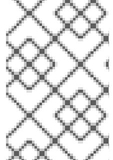
B.3. CUSTOMIZED KARAF ASSEMBLY

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's **pom.xml** file. Both the bundles (or features) that are specified in **karaf-maven-plugin** configuration and the dependencies specified in the **<dependencies>** section in the **pom.xml** can go into the customized karaf assembly.

- for kar
  Dependencies with **kar** type will be added as startup (scope=compile), boot (scope=runtime) or installed (scope=provided) kars in karaf-maven-plugin. The kars are unzipped to working directory (target/assembly) and feature XMLs are searched for and used as additional feature repositories (with stage equal to the stage of given kar).
- for features.xml
  Dependencies with **features** classifier will be used as startup (scope=compile), boot (scope=runtime) or installed (scope=provided) repositories in karaf-maven-plugin. There's no need to explicitly add feature repositories that are found in kar.
- for jar and bundle
  Dependencies with **bundle** or **jar** type will be used as startup (scope=compile), boot (scope=runtime) or installed (scope=provided) bundles in karaf-maven-plugin.

B.3.1. karaf:assembly goal

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 Fuse on OpenShift 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 **bootBundles** element.



NOTE

Include only the necessary elements like karaf framework feature in startup stage as it will go into **etc/startup.properties** and at this stage karaf features service is not fully started. Defer other elements to boot stage.

Example

The following example displays the typical Maven configuration in a quickstart:

```
<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>karaf-maven-plugin</artifactId>
  <version>${fuse.version}</version>
  <extensions>true</extensions>
  <executions>
    <execution>
      <id>karaf-assembly</id>
      <goals>
        <goal>assembly</goal>
      </goals>
      <phase>install</phase>
    </execution>
  </executions>
  <configuration>

    <karafVersion>{karafMavenPluginVersion}</karafVersion>
    <useReferenceUrls>true</useReferenceUrls>
    <archiveTarGz>false</archiveTarGz>
    <includeBuildOutputDirectory>false</includeBuildOutputDirectory>
    <startupFeatures>
      <feature>karaf-framework</feature>
          </startupFeatures>
          <bootFeatures>
      <feature>shell</feature>
      <feature>jaas</feature>
      <feature>aries-blueprint</feature>
      <feature>camel-blueprint</feature>
      <feature>fabric8-karaf-blueprint</feature>
      <feature>fabric8-karaf-checks</feature>
    </bootFeatures>
    <bootBundles>
      <bundle>mvn:${project.groupId}/${project.artifactId}/${project.version}</bundle>
    </bootBundles>
  </configuration>
</plugin>
```

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:

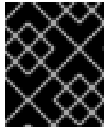
- Building Docker-formatted images and,
- Creating OpenShift resource descriptors

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 on OpenShift 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 four 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.

*Docker Compose is used to bring up docker compose deployments on a OpenShift cluster. This requires minimum to no knowledge of OpenShift deployment process. 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>org.jboss.redhat-fuse</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${fuse.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.

C.3.1. Understanding Build and Development Goals:

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 on OpenShift 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
fabric8:watch	Monitor the project workspace for changes and automatically trigger redeployment of application.

C.3.2. Setting Environmental Variable

You can set one or more environment variables by adding the env parameter in the XML configuration. For example,

```
<configuration>
  <resources>
    <env>
        <JAVA_OPTIONS>-Dmy.custom=option</JAVA_OPTIONS>
        <MY_VAR>value</MY_VAR>
```

```
    </env>
  </resources>
</configuration>
```

C.3.3. Resource Validation Configuration

The **fabric8:resource** goal validates the generated resource descriptors using API specification of Kubernetes and OpenShift.

Table C.3. Resource Validation Configuration

Configuration	Description	Default
fabric8.skipResourceValidation	If value is set to true then resource validation is skipped. This is useful when the resource validation is failing for some reason but you still want to continue the deployment.	false
fabric8.failOnValidationError	If value is set to true then any validation error will block the plugin execution. A warning will be displayed otherwise.	false
fabric8.build.switchToDeployment	If value is set to true then fabric8-maven-plugin would switch to Deployments rather than DeploymentConfig when not using ImageStreams on OpenShift.	false
fabric8.openshift.trimImageInContainerSpec	If value is set to true then it would set the container image reference to "", this is done to handle weird behavior of OpenShift 3.7 in which subsequent rollouts lead to **ImagePullErr**.	false

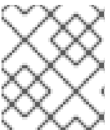
For more information about the Fabric8 Maven plugin goals, see https://maven.fabric8.io/#goals.

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 on OpenShift, 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 on OpenShift 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 on OpenShift, 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>
```

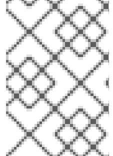
Where **<namespace>** is the name of the OpenShift project where the image streams are defined (normally, **openshift**), **<image-stream-name>** is the name of the image stream, and **<tag>** identifies a particular image in the stream (or tracks the *latest* image in the stream).

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>**.



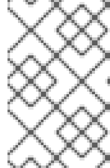
NOTE

The default behavior 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>**):

```
fuse7/fuse-eap-openshift:1.3
fuse7/fuse-java-openshift:1.3
fuse7/fuse-karaf-openshift:1.3
```



NOTE

In the Fuse image streams, the individual images are tagged with build numbers — for example, **1.0-1**, **1.0-2**, and so on. The **1.0** 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.redhat.io**, the Fabric8 Maven plug-in uses the following default image specifiers to select the Fuse images (formatted as **<image-namespace>/<image-name>:<tag>**):

```
fuse7/fuse-eap-openshift:1.3
fuse7/fuse-java-openshift:1.3
fuse7/fuse-karaf-openshift:1.3
```

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.4. 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.5. 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**
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>2.3.90</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:2.3.80: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:2.3.80: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:2.3.80: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
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.

Parameter	Default Value	Description
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:2.3.80: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 Fuse on OpenShift images. All the options for the Fuse on OpenShift 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.redhat.io/fuse7/fuse-karaf-openshift:1.3 <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.

APPENDIX F. TUNING JVMS TO RUN IN LINUX CONTAINERS

F.1. OVERVIEW

Java processes running inside the Linux container do not behave as expected when you allow JVM ergonomics to set the default values for the garbage collector, heap size, and runtime compiler. When you execute a Java application without any tuning parameters – for example, **java -jar mypplication-fat.jar** – the JVM automatically sets several parameters based on the host limits, *not* the container limits.

This section provides information about the packaging of Java applications inside a Linux container so that the container's limits are taken into consideration for calculating default values.

F.2. TUNING THE JVM

The current generation of Java JVMs are not container-aware, so they allocate resources based on the size of the physical host, not on the size of the container. For example, a JVM normally sets the *maximum heap size* to be 1/4 of the physical memory on a host. On a large host machine, this value can easily exceed the memory limit defined for a container and, if the container limit is exceeded at run time, OpenShift will kill the application.

To solve this issue, you can use the Fuse on OpenShift base image that is capable of understanding that a Java JVM runs inside a restricted container and automatically adjusts the maximum heap size, if not done manually. It provides a solution of setting the maximum memory limit and the core limit on the JVM that runs your application. For Fuse on OpenShift images, it can:

- Set CICompilerCount based on the container cores
- Disable C2 JIT compiler when container memory limit is below 300MB
- Use one-fourth of the container memory limit for the default heap size when below 300MB

F.3. DEFAULT BEHAVIOUR OF FUSE ON OPENSHIFT IMAGES

In Fuse on OpenShift, the base image for an application build can either be a Java image (for Spring Boot applications) or a Karaf image (for Karaf applications). Fuse on OpenShift images execute a script that reads the container limits and uses these limits as the basis for allocating resources. By default, the script allocates the following resources to the JVM:

- 50% of the container memory limit,
- 50% of the container core limit.

There are some exceptions to this. For Karaf and Java images, when the physical memory is below 300MB threshold, heap size is restored to one-fourth default heap size instead of the one-half.

F.4. CUSTOM TUNING OF FUSE ON OPENSHIFT IMAGES

The script sets the **CONTAINER_MAX_MEMORY** and **CONTAINER_CORE_LIMIT** environment variables, which can be read by a custom application to tune its internal resources. Additionally, you can specify the following runtime environment variables that enable you to customize the settings on the JVM that runs your application:

- **JAVA_OPTIONS**

- **JAVA_MAX_MEM_RATIO**

To customize the limits explicitly, you can set the **JAVA_MAX_MEM_RATIO** environment variable by editing the **deployment.yml** file, in your Maven project. For example:

```
spec:
  template:
    spec:
      containers:
        -
          resources:
            requests:
              cpu: "0.2"
               memory: 256Mi
            limits:
              cpu: "1.0"
               memory: 256Mi
          env:
          - name: JAVA_MAX_MEM_RATIO
            value: 60
```

F.5. TUNING THIRD-PARTY LIBRARIES

Red Hat recommends you to customize limits for any third-party Java libraries such as Jetty. These libraries would use the given default limits, if you fail to customize limits manually.

The startup script exposes some environment variables describing container limits which can be used by applications:

CONTAINER_CORE_LIMIT

A calculated core limit

CONTAINER_MAX_MEMORY

Memory limit given to the container