



Red Hat Fuse 7.5

Fuse on OpenShift Guide

Installing and developing with Red Hat Fuse on OpenShift

Red Hat Fuse 7.5 Fuse on OpenShift Guide

Installing and developing with Red Hat Fuse on OpenShift

Legal Notice

Copyright © 2023 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 OPENSIFT | 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 OPENSIFT 4.X SERVER | 10 |
| 2.2.1. Setting up the Fuse Console on OpenShift 4.x | 13 |
| 2.2.1.1. Generating a certificate to secure the Fuse Console on OpenShift 4.x | 13 |
| 2.2.1.2. Installing and deploying the Fuse Console on OpenShift 4.x by using the command line | 15 |
| 2.2.1.3. Upgrading the Fuse Console on OpenShift 4.x | 17 |
| 2.2.2. Installing API Designer on OpenShift 4.x | 18 |
| 2.2.2.1. Adding API Designer as a service to an OpenShift 4.x project | 19 |
| 2.2.2.2. Upgrading the API Designer on OpenShift 4.x | 20 |
| 2.3. INSTALLING FUSE IMAGESTREAMS AND TEMPLATES ON THE OPENSIFT 3.X SERVER | 20 |
| 2.3.1. Set up the Fuse Console on OpenShift 3.11 | 22 |
| 2.3.1.1. Deploying the Fuse Console from the OpenShift 3.11 Console | 22 |
| 2.3.1.2. Monitoring a single Fuse pod from the Fuse Console on OpenShift 3.11 | 23 |
| 2.3.1.3. Deploying the Fuse Console from the command line | 25 |
| CHAPTER 3. INSTALLING FUSE ON OPENSIFT AS A NON-ADMIN USER | 27 |
| 3.1. INSTALLING FUSE ON OPENSIFT IMAGES AND TEMPLATES AS A NON-ADMIN USER | 27 |
| CHAPTER 4. GETTING STARTED FOR DEVELOPERS | 29 |
| 4.1. PREPARING DEVELOPMENT ENVIRONMENT | 29 |
| 4.1.1. Installing Container Development Kit (CDK) on your local machine | 29 |
| 4.1.2. Getting remote access to an existing OpenShift server | 30 |
| 4.1.3. Installing Client-Side tools | 30 |
| 4.1.4. Configuring Maven repositories | 31 |
| 4.2. CREATING AND DEPLOYING APPLICATIONS ON FUSE ON OPENSIFT | 31 |
| 4.2.1. Creating and deploying an application using the S2I binary workflow | 31 |
| 4.2.2. Undeploying and redeploying the project | 35 |
| 4.2.3. Creating and deploying an application using the S2I source workflow | 35 |
| CHAPTER 5. DEVELOPING AN APPLICATION FOR THE SPRING BOOT IMAGE | 39 |
| 5.1. CREATING A SPRING BOOT PROJECT USING MAVEN ARCHETYPE | 39 |
| 5.2. CREATING A SPRING BOOT 2 PROJECT USING MAVEN ARCHETYPE | 40 |
| 5.3. STRUCTURE OF THE CAMEL SPRING BOOT APPLICATION | 40 |
| 5.4. SPRING BOOT ARCHETYPE CATALOG | 43 |
| 5.5. SPRING BOOT 2 ARCHETYPE CATALOG | 44 |
| 5.6. BOM FILE FOR SPRING BOOT | 45 |
| 5.6.1. Incorporate the BOM file | 46 |
| 5.7. SPRING BOOT MAVEN PLUGIN | 47 |
| CHAPTER 6. RUNNING APACHE CAMEL APPLICATION IN SPRING BOOT | 49 |
| 6.1. INTRODUCTION TO THE CAMEL SPRING BOOT COMPONENT | 49 |
| 6.2. INTRODUCTION TO THE CAMEL SPRING BOOT STARTER MODULE | 49 |
| 6.3. LIST OF THE CAMEL COMPONENTS THAT DO NOT HAVE STARTER MODULES | 50 |
| 6.4. USING CAMEL SPRING BOOT STARTER | 50 |
| 6.5. ABOUT CAMEL CONTEXT AUTO-CONFIGURATION FOR SPRING BOOT | 51 |
| 6.6. AUTO-DETECTING CAMEL ROUTES IN SPRING BOOT APPLICATIONS | 52 |
| 6.7. CONFIGURING CAMEL PROPERTIES FOR CAMEL SPRING BOOT AUTO-CONFIGURATION | 53 |

| | |
|---|-----------|
| 6.8. CONFIGURING CUSTOM CAMEL CONTEXT | 53 |
| 6.9. DISABLING JMX IN THE AUTO-CONFIGURED CAMELCONTEXT | 54 |
| 6.10. INJECTING AUTO-CONFIGURED CONSUMER AND PRODUCER TEMPLATES INTO SPRING-MANAGED BEANS | 54 |
| 6.11. ABOUT THE AUTO-CONFIGURED TYPECONVERTER IN THE SPRING CONTEXT | 54 |
| 6.12. SPRING TYPE CONVERSION API BRIDGE | 55 |
| 6.13. DISABLING TYPE CONVERSIONS FEATURES | 55 |
| 6.14. ADDING XML ROUTES TO THE CLASSPATH FOR AUTO-CONFIGURATION | 56 |
| 6.15. ADDING XML REST-DSL ROUTES FOR AUTO-CONFIGURATION | 56 |
| 6.16. TESTING WITH CAMEL SPRING BOOT | 57 |
| CHAPTER 7. RUNNING A CAMEL SERVICE ON SPRING BOOT WITH XA TRANSACTIONS | 59 |
| 7.1. STATEFULSET RESOURCES | 59 |
| 7.2. SPRING BOOT NARAYANA RECOVERY CONTROLLER | 59 |
| 7.3. CONFIGURING SPRING BOOT NARAYANA RECOVERY CONTROLLER | 59 |
| 7.4. RUNNING CAMEL SPRING BOOT XA QUICKSTART ON OPENSIFT | 60 |
| 7.5. TESTING SUCCESSFUL XA TRANSACTIONS | 62 |
| 7.6. TESTING FAILED XA TRANSACTIONS | 62 |
| CHAPTER 8. INTEGRATING A CAMEL APPLICATION WITH THE A-MQ BROKER | 63 |
| 8.1. BUILDING AND DEPLOYING A SPRING BOOT CAMEL A-MQ QUICKSTART | 63 |
| CHAPTER 9. INTEGRATING SPRING BOOT WITH KUBERNETES | 66 |
| 9.1. SPRING BOOT EXTERNALIZED CONFIGURATION | 66 |
| 9.1.1. Kubernetes ConfigMap | 66 |
| 9.1.2. Kubernetes Secrets | 66 |
| 9.1.3. Spring Cloud Kubernetes plugin | 66 |
| 9.1.4. Enabling Spring Boot with Kubernetes integration | 66 |
| 9.2. RUNNING TUTORIAL FOR CONFIGMAP PROPERTY SOURCE | 67 |
| 9.2.1. Running Spring Boot Camel Config quickstart | 67 |
| 9.2.2. Configuration properties bean | 69 |
| 9.2.3. Setting up Secret | 71 |
| 9.2.4. Setting up ConfigMap | 73 |
| 9.3. USING CONFIGMAP PROPERTYSOURCE | 75 |
| 9.3.1. Applying individual properties | 75 |
| 9.3.2. Applying application.yaml ConfigMap property | 75 |
| 9.3.3. Applying application.properties ConfigMap property | 75 |
| 9.3.4. Deploying a ConfigMap | 76 |
| 9.4. USING SECRETS PROPERTYSOURCE | 76 |
| 9.4.1. Example of setting Secrets | 76 |
| 9.4.2. Consuming the Secrets | 77 |
| 9.4.3. Configuration properties for Secrets PropertySource | 78 |
| 9.5. USING PROPERTYSOURCE RELOAD | 78 |
| 9.5.1. Enabling PropertySource Reload | 78 |
| 9.5.2. Levels of PropertySource Reload | 78 |
| 9.5.3. Example of PropertySource Reload | 79 |
| 9.5.4. PropertySource Reload operating modes | 80 |
| 9.5.5. PropertySource Reload configuration properties | 80 |
| CHAPTER 10. DEVELOPING AN APPLICATION FOR THE KARAF IMAGE | 81 |
| 10.1. CREATING A KARAF PROJECT USING MAVEN ARCHETYPE | 81 |
| 10.2. STRUCTURE OF THE CAMEL KARAF APPLICATION | 82 |
| 10.3. KARAF ARCHETYPE CATALOG | 83 |
| 10.4. USING FABRIC8 KARAF FEATURES | 83 |

| | |
|--|------------|
| 10.4.1. Adding Fabric8 Karaf features | 83 |
| 10.4.2. Adding Fabric8 Karaf Core bundle functionality | 84 |
| 10.4.3. Setting the Property Placeholder service options | 84 |
| 10.4.4. Adding a custom property placeholder resolver | 86 |
| 10.4.5. List of resolution strategies | 87 |
| 10.4.6. List of Property Placeholder service options | 87 |
| 10.5. ADDING FABRIC8 KARAF CONFIG ADMIN SUPPORT | 88 |
| 10.5.1. Adding Fabric8 Karaf Config admin support | 88 |
| 10.5.2. Adding ConfigMap injection | 88 |
| 10.5.3. Configuration plugin | 89 |
| 10.5.4. Config Property Placeholders | 89 |
| 10.5.5. Fabric8 Karaf Config Admin options | 89 |
| 10.6. ADDING FABRIC8 KARAF BLUEPRINT SUPPORT | 90 |
| 10.7. ENABLING FABRIC8 KARAF HEALTH CHECKS | 91 |
| 10.8. ADDING CUSTOM HEALTH CHECKS | 92 |
| CHAPTER 11. DEVELOPING AN APPLICATION FOR THE JBOSS EAP IMAGE | 94 |
| 11.1. CREATING A JBOSS EAP PROJECT USING THE S2I SOURCE WORKFLOW | 94 |
| 11.2. STRUCTURE OF THE JBOSS EAP APPLICATION | 97 |
| 11.3. JBOSS EAP QUICKSTART TEMPLATES | 97 |
| CHAPTER 12. USING PERSISTENT STORAGE IN FUSE ON OPENSIFT | 99 |
| 12.1. ABOUT VOLUMES AND VOLUME TYPES | 99 |
| 12.2. ABOUT PERSISTENTVOLUMES | 99 |
| 12.3. CONFIGURING PERSISTENT VOLUME | 99 |
| 12.4. CREATING PERSISTENTVOLUMECLAIMS | 100 |
| 12.5. USING PERSISTENT VOLUMES IN PODS | 100 |
| CHAPTER 13. PATCHING FUSE ON OPENSIFT | 102 |
| 13.1. IMPORTANT NOTE ON BOMS AND MAVEN DEPENDENCIES | 102 |
| 13.2. PATCHING THE FUSE ON OPENSIFT IMAGES | 102 |
| 13.3. PATCHING THE FUSE ON OPENSIFT TEMPLATES | 103 |
| 13.4. PATCH APPLICATION DEPENDENCIES USING BOM | 104 |
| 13.4.1. Updating dependencies in a Spring Boot application | 104 |
| 13.4.2. Updating dependencies in a Karaf application | 106 |
| 13.4.3. Updating dependencies in a JBoss EAP application | 107 |
| 13.5. AVAILABLE BOM VERSIONS | 107 |
| APPENDIX A. SPRING BOOT MAVEN PLUGIN | 108 |
| A.1. SPRING BOOT MAVEN PLUGIN GOALS | 108 |
| A.2. USING SPRING BOOT MAVEN PLUGIN | 108 |
| A.2.1. Using Spring Boot Maven plugin for Spring Boot 2 | 108 |
| A.2.2. Using Spring Boot Maven plugin for Spring Boot 1 | 110 |
| APPENDIX B. USING KARAF MAVEN PLUGIN | 112 |
| B.1. MAVEN DEPENDENCIES | 112 |
| B.2. KARAF MAVEN PLUGIN CONFIGURATION | 112 |
| B.3. CUSTOMIZED KARAF ASSEMBLY | 113 |
| B.3.1. karaf:assembly goal | 113 |
| APPENDIX C. FABRIC8 MAVEN PLUGIN | 115 |
| C.1. BUILDING IMAGES | 115 |
| C.2. KUBERNETES AND OPENSIFT RESOURCES | 115 |
| C.3. INSTALLING THE PLUGIN | 116 |
| C.4. UNDERSTANDING FABRIC8 MAVEN PLUGIN GOALS | 116 |

| | |
|---|------------|
| C.4.1. Understanding build and development goals | 116 |
| C.4.2. Setting environmental variable | 117 |
| C.4.3. Resource validation configuration | 118 |
| C.5. GENERATORS | 118 |
| C.5.1. Zero configuration | 118 |
| C.5.2. Modes for specifying the base image | 119 |
| C.5.2.1. Default values for istag mode | 119 |
| C.5.2.2. Default values for docker mode | 120 |
| C.5.2.3. Mode configuration for Spring Boot applications | 120 |
| C.5.2.4. Mode configuration for Karaf applications | 120 |
| C.5.2.5. Specifying the Generator mode using the command line | 120 |
| C.5.3. Spring Boot | 121 |
| C.5.4. Karaf | 121 |
| APPENDIX D. FABRIC8 CAMEL MAVEN PLUGIN | 123 |
| D.1. FABRIC8 CAMEL MAVEN PLUGIN GOALS | 123 |
| D.2. ADDING THE FABRIC8-CAMEL-MAVEN PLUGIN TO YOUR PROJECT | 123 |
| D.3. RUNNING THE GOAL ON ANY MAVEN PROJECT | 124 |
| D.4. OPTIONS | 125 |
| D.5. VALIDATING INCLUDE TEST | 126 |
| APPENDIX E. CUSTOMIZING JVM ENVIRONMENT VARIABLES | 128 |
| E.1. USING S2I JAVA BUILDER IMAGE WITH OPENJDK 8 | 128 |
| E.2. USING S2I KARAF BUILDER IMAGE WITH OPENJDK 8 | 128 |
| E.2.1. Configuring the Karaf4 assembly | 128 |
| E.2.2. Customizing the Maven build | 128 |
| E.3. BUILD TIME ENVIRONMENT VARIABLES | 128 |
| E.4. RUN TIME ENVIRONMENT VARIABLES | 129 |
| E.5. JOLOKIA CONFIGURATION | 129 |
| APPENDIX F. TUNING JVMs TO RUN IN LINUX CONTAINERS | 131 |
| F.1. TUNING THE JVM | 131 |
| F.2. DEFAULT BEHAVIOUR OF FUSE ON OPENSIFT IMAGES | 131 |
| F.3. CUSTOM TUNING OF FUSE ON OPENSIFT IMAGES | 131 |
| F.4. TUNING THIRD-PARTY LIBRARIES | 132 |

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 OPENSIFT

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:

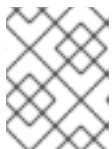
```
oc login -u system:admin
```

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 OPENSIFT 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
```

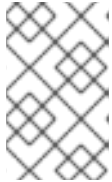
6. Add the Fuse imagestreams Skipped Imagestreams section and add Fuse and Spring Boot 2 templates to Skipped Templates section.

```
[...]
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-fuse75-eap-camel-amq
  - s2i-fuse75-eap-camel-cdi
  - s2i-fuse75-eap-camel-cxf-jaxrs
  - s2i-fuse75-eap-camel-cxf-jaxws
  - s2i-fuse75-eap-camel-jpa
  - s2i-fuse75-karaf-camel-amq
  - s2i-fuse75-karaf-camel-log
  - s2i-fuse75-karaf-camel-rest-sql
  - s2i-fuse75-karaf-cxf-rest
  - s2i-fuse75-spring-boot-camel
  - s2i-fuse75-spring-boot-camel-amq
  - s2i-fuse75-spring-boot-camel-config
  - s2i-fuse75-spring-boot-camel-drools
  - s2i-fuse75-spring-boot-camel-infinispan
  - s2i-fuse75-spring-boot-camel-rest-sql
  - s2i-fuse75-spring-boot-camel-rest-3scale
  - s2i-fuse75-spring-boot-camel-xa
  - s2i-fuse75-spring-boot-camel-xml
  - s2i-fuse75-spring-boot-cxf-jaxrs
  - s2i-fuse75-spring-boot-2-camel-amq
  - s2i-fuse75-spring-boot-2-camel-drools
  - s2i-fuse75-spring-boot-2-camel-infinispan
  - s2i-fuse75-spring-boot-2-camel-rest-3scale
  - s2i-fuse75-spring-boot-2-camel-rest-sql
  - s2i-fuse75-spring-boot-2-camel-teiid
  - s2i-fuse75-spring-boot-2-camel
  - s2i-fuse75-spring-boot-2-camel-xa
  - s2i-fuse75-spring-boot-2-camel-xml
  - s2i-fuse75-spring-boot-2-cxf-jaxrs
  - s2i-fuse75-spring-boot-2-cxf-jaxws
```

- 7. Install Fuse on OpenShift image streams.

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-750056-redhat-00006
```

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



NOTE

If an error is displayed, with the message "Error from server (AlreadyExists): imagestreams.image.openshift.io <imagestreamname> already exists", use the following command to replace the existing imagestreams with the latest.

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

- 8. Install Fuse on OpenShift 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-rest-3scale-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-750056-redhat-00006/quickstarts/${template}
done
```

- 9. Install Spring Boot 2 quickstart templates:

```
for template in spring-boot-2-camel-amq-template.json \
  spring-boot-2-camel-drools-template.json \
  spring-boot-2-camel-infinispan-template.json \
  spring-boot-2-camel-rest-3scale-template.json \
  spring-boot-2-camel-rest-sql-template.json \
  spring-boot-2-camel-teiid-template.json \
  spring-boot-2-camel-template.json \
```



```

spring-boot-2-camel-xa-template.json \
spring-boot-2-camel-xml-template.json \
spring-boot-2-cxf-jaxrs-template.json \
spring-boot-2-cxf-jaxws-template.json ;
do oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-
2.1.fuse-sb2-750016-redhat-00005/quickstarts/${template}
done

```

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

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

2.2.1. Setting up the Fuse Console on OpenShift 4.x

On OpenShift 4.x, setting up the Fuse Console involves securing, installing, and deploying it.

First, you must generate a client certificate so that you can secure the Fuse Console as described in [Section 2.2.1.1, “Generating a certificate to secure the Fuse Console on OpenShift 4.x”](#) .

After you generate the client certificate, you can use the command line and one of the Fuse Console templates to install and deploy the Fuse Console so that it has access to Fuse applications in multiple namespaces on the OpenShift cluster or in a specific namespace.



NOTE

- User management for the Fuse Console is handled by OpenShift.
- Role-based access control (for users accessing the Fuse Console after it is deployed) is not yet available for Fuse on OpenShift.

2.2.1.1. Generating a certificate to secure the Fuse Console on OpenShift 4.x

On OpenShift 4.x, to keep the connection between the Fuse Console proxy and the Jolokia agent secure, you must generate a client certificate before you deploy the Fuse Console. You must use the service signing certificate authority private key to sign the client certificate.



IMPORTANT

You must generate and sign a separate client certificate for each OpenShift cluster. Do not use the same certificate for more than one cluster.

Prerequisites

- You have **cluster admin** access to the OpenShift cluster.
- If you are generating certificates for more than one OpenShift cluster and you previously generated a certificate for a different cluster in the current directory, do one of the following to ensure that you generate a different certificate for the current cluster:
 - Delete the existing certificate files (for example, **ca.crt**, **ca.key**, and **ca.srl**) from the current directory.

- Change to a different working directory. For example, if your current working directory is named **cluster1**, create a new **cluster2** directory and change your working directory to it:

```
mkdir ../cluster2
```

```
cd ../cluster2
```

Procedure

1. Login to OpenShift as a user with cluster admin access:

```
oc login -u <user_with_cluster_admin_role>
```

2. Retrieve the service signing certificate authority keys, by executing the following commands:

- To retrieve the certificate:

```
oc get secrets/signing-key -n openshift-service-ca -o "jsonpath={.data['tls.crt']}" | base64 --decode > ca.crt
```

- To retrieve the private key:

```
oc get secrets/signing-key -n openshift-service-ca -o "jsonpath={.data['tls.key']}" | base64 --decode > ca.key
```

3. Generate the client certificate, as documented in [Kubernetes certificates administration](#), using either **easysrsa**, **openssl**, or **cfssl**.

Here are the example commands using openssl:

- a. Generate the private key:

```
openssl genrsa -out server.key 2048
```

- b. Write the CSR config file.

```
cat <<EOT >> csr.conf
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
distinguished_name = dn

[ dn ]
CN = fuse-console.fuse.svc

[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
keyUsage=keyEncipherment,dataEncipherment,digitalSignature
extendedKeyUsage=serverAuth,clientAuth
EOT
```

- c. Generate the CSR:

```
openssl req -new -key server.key -out server.csr -config csr.conf
```

d. Issue the signed certificate:

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out server.crt
-days 10000 -extensions v3_ext -extfile csr.conf
```

Next steps

You need this certificate to create the secret for the Fuse Console as described in [Installing and deploying the Fuse Console on OpenShift 4.x by using the command line](#).

2.2.1.2. Installing and deploying the Fuse Console on OpenShift 4.x by using the command line

On OpenShift 4.x, you can choose one of these deployment options to install and deploy the Fuse Console from the command line:

- **cluster** - The Fuse Console can discover and connect to Fuse applications deployed across multiple namespaces (projects) on the OpenShift cluster. To deploy this template, you must have the administrator role for the OpenShift cluster.
- **namespace** - The Fuse Console has access to a specific OpenShift project (namespace). To deploy this template, you must have the administrator role for the OpenShift project.

To view a list of the parameters for the Fuse Console templates, run the following OpenShift command:

```
oc process --parameters -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-750056-redhat-00006/fuse-console-namespace-os4.json
```

Prerequisites

- Before you install and deploy the Fuse Console, you must generate a client certificate that is signed with the service signing certificate authority as described in [Generating a certificate to secure the Fuse Console on OpenShift 4.x](#).
- You have the **cluster admin** role for the OpenShift cluster.
- The Fuse Console image stream (along with the other Fuse image streams) are installed, as described in [Installing Fuse imagestreams and templates on the OpenShift 4.x server](#).

Procedure

1. Verify that the Fuse Console image stream is installed by using the following command to retrieve a list of all templates:

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

2. Optionally, if you want to update the already installed image stream with new release tags, use the following command to import the Fuse Console image to the **openshift** namespace:

```
oc import-image fuse7/fuse7-console:1.5 --from=registry.redhat.io/fuse7/fuse-console:1.5 --confirm -n openshift
```

3. Download one of the following Fuse Console templates from <https://raw.githubusercontent.com/jboss-fuse/application-templates/application->

[templates-2.1.fuse-750056-redhat-00006](#) in raw format to a local directory of your choice (for example, `mytemp`).

- Cluster template: **fuse-console-cluster-os4.json**
 - Namespace template: **fuse-console-namespace-os4.json**
4. For the namespace template, skip to the next step.
For the cluster template, in an editor of your choice, edit your local copy of the **fuse-console-cluster-os4.json** file to add the **"grantMethod": "auto"** attribute to the **OAuthClient** section:

```
{
  "kind": "OAuthClient",
  "apiVersion": "v1",
  "metadata": {
    "name": "${APP_NAME}-oauth-client"
  },
  "grantMethod": "auto",
  "redirectURIs": [
    "https://${ROUTE_HOSTNAME}"
  ]
}
```

5. Obtain the Fuse Console **APP_NAME** value by running the following command (where **mytemp** is the path to the local directory that contains the Fuse Console template):

- For the cluster template:

```
oc process --parameters -f mytemp/fuse-console-cluster-os4.json
```

- For the namespace template:

```
oc process --parameters -f mytemp/fuse-console-namespace-os4.json
```

6. From the certificate that you generated in [Securing the Fuse Console on OpenShift 4.x](#), create the secret and mount it in the Fuse Console by using the following command (where **APP_NAME** is the name of the Fuse Console application).

```
oc create secret tls APP_NAME-tls-proxying --cert server.crt --key server.key
```

7. Create a new application based on your local copy of the Fuse Console template by running the following command (where **myproject** is the name of your OpenShift project, **mytemp** is the path to the local directory that contains the Fuse Console template, and **myhost** is the hostname to access the Fuse Console):

- For the cluster template:

```
oc new-app -n myproject -f mytemp/fuse-console-cluster-os4.json -p
ROUTE_HOSTNAME=myhost
```

- For the namespace template:

```
oc new-app -n myproject -f mytemp/fuse-console-namespace-os4.json
```

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

```
oc status
```

- To access the Fuse Console from a browser, use the URL that is returned in Step 6 (for example, <https://fuse-console.192.168.64.12.nip.io>).

2.2.1.3. Upgrading the Fuse Console on OpenShift 4.x

Red Hat OpenShift 4.x handles updates to operators, including the Red Hat Fuse operators. For more information see the [Operators OpenShift documentation](#).

In turn, operator updates can trigger application upgrades. How an application upgrade occur differs according to how the application is configured.

For Fuse Console applications, you can specify when to trigger upgrades to those applications by editing the **.spec.version** field of the application custom resource definition.

Prerequisite

- You have OpenShift cluster admin permissions.

Procedure

To upgrade a Fuse Console application:

- In a terminal window, use the following command to change the **.spec.version** field of the application custom resource definition:

```
oc patch <project-name> <custom-resource-name> --type='merge' -p '{"spec": {"version": "1.7.1"}}'
```

For example:

```
oc patch myproject example-fuseconsole --type='merge' -p '{"spec":{"version":"1.7.1"}}'
```

- Check that the application's status has updated:

```
oc get myproject
```

The response shows information about the application, including the version number:

| NAME | AGE | URL | IMAGE |
|---------------------|-----|---|------------------------------------|
| example-fuseconsole | 1m | https://fuseconsole.192.168.64.38.nip.io | docker.io/fuseconsole/online:1.7.1 |

When you change the value of the **.spec.version** field, OpenShift automatically redeploys the application.

- To check the status of the redeployment that is triggered by the version change:

```
oc rollout status deployment.v1.apps/example-fuseconsole
```

A successful deployment shows this response:

deployment "example-fuseconsole" successfully rolled out

2.2.2. Installing API Designer on OpenShift 4.x

Red Hat Fuse on OpenShift provides API Designer, a web-based API designer tool that you can use to design REST APIs. The API Designer operator simplifies the installation and upgrading of API Designer on OpenShift Container Platform 4.x.

As an OpenShift administrator, you install the API Designer operator to an OpenShift project (namespace). When the operator is installed, the operator is running in the selected namespace. However, to make the API Designer available as a service, either you, as the OpenShift administrator, or a developer must create an instance of the API Designer. The API Designer service provides the URL to access the API Designer web console.

Prerequisites

- You have administrator access to the OpenShift cluster.
- You have configured authentication to the Red Hat Container Registry.

Procedure

1. Start the OpenShift 4.x Server.
2. In a web browser, navigate to the OpenShift console in your browser. Log in to the console with your credentials.
3. Click **Catalog** and then click **OperatorHub**.
4. In the search field, type **API Designer**.
You can see the API Designer card in the right-side panel.
5. Click **API Designer**. The API Designer operator install window appears.
6. Click **Install**. The **Create Operator Subscription** form opens.
 - a. For **Installation mode**, select a namespace (project) from the list of namespaces on the cluster,
 - b. For the **Approval Strategy**, select **Automatic** or **Manual** to configure how OpenShift handles updates to the API Designer Operator.
 - If you select **Automatic** updates, when a new version of the API Designer operator is available, the OpenShift Operator Lifecycle Manager (OLM) automatically upgrades the running instance of the API Designer without human intervention.
 - If you select **Manual** updates, when a newer version of an Operator is available, the OLM creates an update request. As a cluster administrator, you must then manually approve that update request to have the API Designer operator updated to the new version.
7. Click **Subscribe** to make the API Designer Operator available to the specified namespace (project).
8. To verify that the API Designer is installed in the project, click **Operators** and then click **Installed Operators** to see the API Designer in the list.

Next Steps

After the API Designer operator is installed, the API Designer must be added as a service to the OpenShift project by creating an instance of the API Designer. This task can be accomplished in two ways:

- An OpenShift administrator can follow the steps in [Section 2.2.2.1, “Adding API Designer as a service to an OpenShift 4.x project”](#).
- An OpenShift developer can follow the steps described in [Designing APIs](#).
The API Designer service provides the URL to access the API Designer web console.

2.2.2.1. Adding API Designer as a service to an OpenShift 4.x project

After the API Designer operator is installed in an OpenShift 4.x project, you (or an OpenShift developer) can add it as a service to the OpenShift project. The API Designer service provides the URL that a developer uses to access the API Designer web console.



NOTE

See [Designing APIs](#) for the steps that an OpenShift developer follows to add API Designer as a service to an OpenShift 4.x project.

Prerequisites

- You have administrator access to the OpenShift cluster.
- The API Designer operator is installed into the current OpenShift project.

Procedure

1. In the OpenShift web console, click **Operators** and then click **Installed Operators**.
2. In the **Name** column, click **API Designer**.
3. Under **Provided APIs**, click **Create Instance**.
A default form with a minimal starting template for the API Designer instance opens.
4. Accept the default values or optionally edit the following values:
 - **size**: The number of pods for the API Designer instance.
The default is three pods for three new resource types that contain logic specific to managing API Designer.
 - **image**: The API Designer image. If you change this image, the API Designer operator is automatically upgraded on the cluster.
5. Click **Create** to create a new **apicurito-service**. OpenShift starts up of the pods, services, and other components for the new API Designer service.
6. To verify that the API Designer service is available:
 - a. Click **Operators** and then click **Installed Operators**.
 - b. In the **Provided APIs** column, click **Apicurito CRD**.
On the **Operator Details** page, the **apicurito-service** is listed.

7. To obtain the URL for the API Designer:
 - a. Click **Networking > Routes**.
 - b. Make sure that the correct project is selected.
 - c. In the **apicurito-service** row, find the URL for the API Designer web console in the **Location** column.

2.2.2.2. Upgrading the API Designer on OpenShift 4.x

Red Hat OpenShift 4.x handles updates to operators, including the Red Hat Fuse operators. For more information see the [Operators OpenShift documentation](#).

In turn, operator updates can trigger application upgrades. How an application upgrade occur differs according to how the application is configured.

For API Designer applications, when you upgrade the API Designer operator, OpenShift automatically also upgrades any API designer applications on the cluster.

2.3. INSTALLING FUSE IMAGESTREAMS AND TEMPLATES ON THE OPENSIFT 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-750056-redhat-00006
```

```
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 \  
do curl -s -L $BASEURL/$template -o $template
```



```

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-rest-3scale-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-750056-redhat-00006/quickstarts/${template}
done

```

6. Install Spring Boot 2 quickstart templates:

```

for template in spring-boot-2-camel-amq-template.json \
spring-boot-2-camel-drools-template.json \
spring-boot-2-camel-infinispan-template.json \
spring-boot-2-camel-rest-3scale-template.json \
spring-boot-2-camel-rest-sql-template.json \
spring-boot-2-camel-teiid-template.json \
spring-boot-2-camel-template.json \
spring-boot-2-camel-xa-template.json \
spring-boot-2-camel-xml-template.json \
spring-boot-2-cxf-jaxrs-template.json \
spring-boot-2-cxf-jaxws-template.json ;
do oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-sb2-750016-redhat-00005/quickstarts/${template}
done

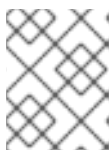
```

7. 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-750056-redhat-00006/fis-console-cluster-template.json
oc create -n openshift -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-750056-redhat-00006/fis-console-namespace-template.json

```



NOTE

For information on deploying the Fuse Console, see [Set up Fuse Console on OpenShift](#).

8. Install the Apicurito template:

```

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

```

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

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

2.3.1. Set up the Fuse Console on OpenShift 3.11

On OpenShift 3.11, you can set up the Fuse Console in two ways:

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

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

Prerequisites

- Install the Fuse on OpenShift image streams and the templates for the Fuse Console as described in [Fuse on OpenShift Guide](#).
- For cluster mode on OpenShift 3.11, 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.
- 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.
- User management for the Fuse Console is handled by OpenShift.
- Role-based access control (for users accessing the Fuse Console after it is deployed) is not yet available for Fuse on OpenShift.

[Section 2.3.1.1, "Deploying the Fuse Console from the OpenShift 3.11 Console"](#)

[Section 2.3.1.2, "Monitoring a single Fuse pod from the Fuse Console on OpenShift 3.11"](#)

[Section 2.3.1.3, "Deploying the Fuse Console from the command line"](#)

2.3.1.1. Deploying the Fuse Console from the OpenShift 3.11 Console

To deploy the Fuse Console on your OpenShift cluster from the OpenShift 3.11 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.

2.3.1.2. Monitoring a single Fuse pod from the Fuse Console on OpenShift 3.11

You can open the Fuse Console for a Fuse pod running on OpenShift 3.11:

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:

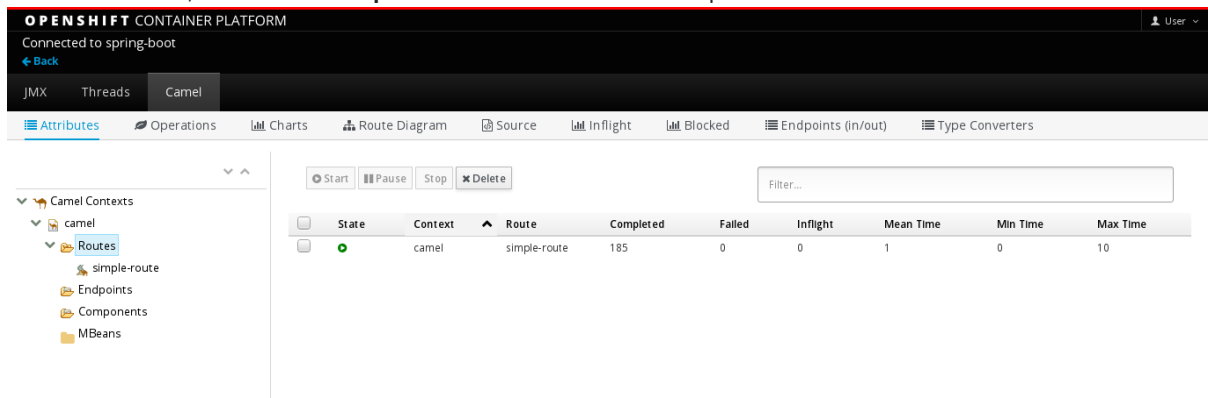
Template

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.



The screenshot shows the OpenShift Container Platform console interface. The top navigation bar includes 'Attributes', 'Operations', 'Charts', 'Route Diagram', 'Source', 'Inflight', 'Blocked', 'Endpoints (in/out)', and 'Type Converters'. The main content area displays a table of Camel routes. The table has columns for State, Context, Route, Completed, Failed, Inflight, Mean Time, Min Time, and Max Time. The 'simple-route' is shown with a green status icon, context 'camel', and 185 completed messages.

| State | Context | Route | Completed | Failed | Inflight | Mean Time | Min Time | Max Time |
|---|---------|--------------|-----------|--------|----------|-----------|----------|----------|
|  | camel | simple-route | 185 | 0 | 0 | 1 | 0 | 10 |



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"
          }
        ]
      }
    ]
  }
}
```

2.3.1.3. Deploying the Fuse Console from the command line

Table 2.1, “Fuse Console templates” describes the OpenShift 3.1 templates that you can use to deploy the Fuse Console from the command line, depending on the type of Fuse application deployment.

Table 2.1. Fuse Console templates

| Type | Description |
|--|--|
| fis-console-cluster-template.json | The Fuse Console can discover and connect to Fuse applications deployed across multiple namespaces or projects. To deploy this template, you must have the OpenShift cluster-admin role. |
| fis-console-namespace-template.json | This template restricts the Fuse Console access to the current OpenShift project (namespace), and as such acts as a single tenant deployment. To deploy this template, you must have the admin role for the current OpenShift project. |

Optionally, you can view a list of the parameters for all of the templates by running this command:

```
oc process --parameters -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.fuse-750056-redhat-00006/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-750056-redhat-00006/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-750056-redhat-00006/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>).

CHAPTER 3. INSTALLING FUSE ON OPENSIFT 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 OPENSIFT 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 3.10 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://OPENSIFT_IP_ADDR:8443
```

Where, **OPENSIFT_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-750056-redhat-00006
```

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

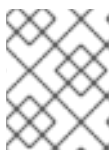
The command output displays the Fuse image streams that are now available in your Fuse on OpenShift project.

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-rest-3scale-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 7.5. 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 3.10 of CDK. Detailed instructions for installing and using CDK 3.10 are provided in the [Red Hat CDK 3.10 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 3.10 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 3.10 *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 3.10 *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 OPENSIFT

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 an application 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-750020-redhat-00002/archetypes-catalog-2.2.0.fuse-750020-redhat-
00002-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-xml-archetype \
-DarchetypeVersion=2.2.0.fuse-750020-redhat-00002
```

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': : fuse75-spring-boot
Define value for property 'version': : 1.0-SNAPSHOT :
Define value for property 'package': : org.example.fis :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse75-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse75-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 **fuse75-spring-boot** subdirectory. You can inspect the XML DSL code in the **fuse75-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. Switch to **openshift** project (if not already in the **openshift** project) as follows.

```
oc project openshift
```

-
- 5. 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](#).

- 6. You are now ready to build and deploy the **fuse75-spring-boot** project. Assuming you are still logged into OpenShift, change to the directory of the **fuse75-spring-boot** project, and then build and deploy the project, as follows.

```
cd fuse75-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: openshift
[INFO] Creating a Service from openshift.yml namespace openshift name fuse75-spring-boot
[INFO] Created Service: target/fabric8/applyJson/openshift/service-fuse75-spring-boot.json
[INFO] Using project: openshift
[INFO] Creating a DeploymentConfig from openshift.yml namespace openshift name fuse75-
spring-boot
[INFO] Created DeploymentConfig: target/fabric8/applyJson/openshift/deploymentconfig-
fuse75-spring-boot.json
[INFO] Creating Route openshift:fuse75-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: 2019-10-24T12:15:06+05:30
[INFO] Final Memory: 63M/688M
[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 left hand side panel, expand **Home** and then click **Status** to view the Project Status page for the **openshift** project.
- 9. Click **fuse75-spring-boot** to view the Overview information page for the **fuse75-spring-boot** application.

Project: openshift ▾ + Add ▾

Project Status

Resources Dashboard

Group by: Application ▾ fuse75

fuse75-spring-boot

DC fuse75-spring-boot, #1 1 of 1 pods

DC fuse75-spring-boot
Actions ▾

Overview Resources

DESIRED COUNT
1 pod

UP-TO-DATE COUNT
1 pod

MATCHING PODS
1 pod

1 available
0 unavailable

| | |
|---|---|
| <p style="font-size: 0.7em; margin: 0;">NAME</p> <p style="margin: 0;">fuse75-spring-boot</p> <p style="font-size: 0.7em; margin: 5px 0 0 0;">NAMESPACE</p> <p style="margin: 0;">NS openshift</p> | <p style="font-size: 0.7em; margin: 0;">LATEST VERSION</p> <p style="margin: 0;">1</p> <p style="font-size: 0.7em; margin: 5px 0 0 0;">REASON</p> <p style="margin: 0;">config change</p> |
|---|---|

10. In the left hand side panel, expand **Workloads**.

11. Click **Pods**. All the running pods in the **openshift** project are displayed.

12. Click on the pod **Name** (in this example, **fuse75-spring-boot-xxxxx**) to view the details of the running pod.

Project: openshift ▾

fuse75-spring-boot-1 > Pod Details

P
fuse75-spring-boot-1-zfgl4

Overview
YAML
Environment
Logs
Events
Terminal

Pod Overview

| | |
|--|---|
| <p style="font-size: 0.7em; margin: 0;">NAME</p> <p style="margin: 0;">fuse75-spring-boot-1-zfgl4</p> <p style="font-size: 0.7em; margin: 5px 0 0 0;">NAMESPACE</p> <p style="margin: 0;">NS openshift</p> <p style="font-size: 0.7em; margin: 5px 0 0 0;">LABELS</p> <p style="margin: 0; font-size: 0.7em;"> app=fuse75-spring-boot deployment=fuse75-spring-boot-1 deploymentconfig=fuse75-spring-boot group=org.example.fis provider=fabric8 version=1.0-SNAPSHOT </p> <p style="font-size: 0.7em; margin: 5px 0 0 0;">TOLERATIONS</p> <p style="margin: 0; font-size: 0.7em;">3 Tolerations ✎</p> | <p style="font-size: 0.7em; margin: 0;">STATUS</p> <p style="margin: 0;">🔄 Running</p> <p style="font-size: 0.7em; margin: 5px 0 0 0;">RESTART POLICY</p> <p style="margin: 0;">Always Restart</p> <p style="font-size: 0.7em; margin: 5px 0 0 0;">POD IP</p> <p style="margin: 0;">10.129.2.15</p> <p style="font-size: 0.7em; margin: 5px 0 0 0;">NODE</p> <p style="margin: 0;">N ip-10-0-161-126.us-east-2.compute.internal</p> |
|--|---|


13. 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.

```

...
06:45:54.311 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 130
06:45:56.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 898
06:45:58.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 414
06:46:00.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 486
06:46:02.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 093
06:46:04.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 080

```

14. To shut down the running pod,

- a. On the Project Status page for the **openshift** project, click **fuse75-spring-boot** application.
- b. Click the **Overview** tab to view to the overview information page of the application.
- c. Click the  icon next to Desired Count. The Edit Count window is displayed.
- d. Use the down arrow to scale down to zero to stop the pod.

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. Creating and deploying an application using the S2I source workflow

In this section, you will use the OpenShift S2I source workflow to build and deploy a Fuse on OpenShift application 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. 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. Switch to **openshift** project (if not already in the **openshift** project) as follows.

```
oc project openshift
```

- Run the following command to ensure that Fuse on OpenShift 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](#).

- Enter the following command to create the resources required for running the **Red Hat Fuse 7.5 Camel XML DSL with Spring Boot** quickstart template. This will create a deployment config and build config for the quickstart. The information about the default parameters of the quickstart and the resources created is displayed on the terminal.

```
oc new-app s2i-fuse75-spring-boot-camel-xml
```

```
--> Deploying template "openshift/s2i-fuse75-spring-boot-camel-xml" to project openshift
...
--> Creating resources ...
  imagestream.image.openshift.io "s2i-fuse75-spring-boot-camel-xml" created
  buildconfig.build.openshift.io "s2i-fuse75-spring-boot-camel-xml" created
  deploymentconfig.apps.openshift.io "s2i-fuse75-spring-boot-camel-xml" created
--> Success
  Build scheduled, use 'oc logs -f bc/s2i-fuse75-spring-boot-camel-xml' to track its progress.
  Run 'oc status' to view your app.
```

- Navigate to the OpenShift web console in your browser (https://OPENSHIFT_IP_ADDR, replace **OPENSHIFT_IP_ADDR** with the IP address of the cluster) and log in to the console with your credentials (for example, with username **developer** and password, **developer**).
- In the left hand side panel, expand **Home**. Click **Status** to view the **Project Status** page. All the existing applications in the selected namespace (for example, openshift) are displayed.
- Click **s2i-fuse75-spring-boot-camel-xml** to view the **Overview** information page for the quickstart.

Project: openshift ▾ + Add ▾

Project Status

Resources Dashboard

Group by: Application ▾ fuse-spring-boot-camel

s2i-fuse75-spring-boot-camel-xml

DC s2i-fuse75-spring-boot-cam... , #1 1 of 1 pods

x

DC s2i-fuse75-spring-boot-camel-xml Actions ▾

Overview Resources

| DESIRED COUNT | UP-TO-DATE COUNT | MATCHING PODS | |
|--|------------------|---------------|------------------------------|
| 1 pod ✎ | 1 pod | 1 pod | } |
| | | | 1 available 0 unavailable |

| NAME | LATEST VERSION |
|----------------------------------|----------------|
| s2i-fuse75-spring-boot-camel-xml | 1 |
| NAMESPACE | REASON |
| NS openshift | config change |

- Click the **Resources** tab and then click **View logs** to view the build log for the application.

[s2i-fuse75-spring-boot-camel-xml](#) > Build Details

B s2i-fuse75-spring-boot-camel-xml-2

Overview [YAML](#) [Environment](#) [Logs](#) [Events](#)

Log stream ended.

```
last 1000 lines
[INFO] Downloading: https://repo1.maven.org/maven2/commons-lang/commons-
[INFO] Downloaded: https://repo1.maven.org/maven2/org/apache/maven/wagon
[INFO] Downloading: https://repo1.maven.org/maven2/org/apache/maven/wagon
[INFO] Downloaded: https://repo1.maven.org/maven2/org/apache/maven/wagon
[INFO] Downloading: https://repo1.maven.org/maven2/org/apache/maven/wagon
[INFO] Downloaded: https://repo1.maven.org/maven2/org/apache/maven/wagon
[INFO] Downloading: https://repo1.maven.org/maven2/org/jsoup/jsoup/1.7.2
```

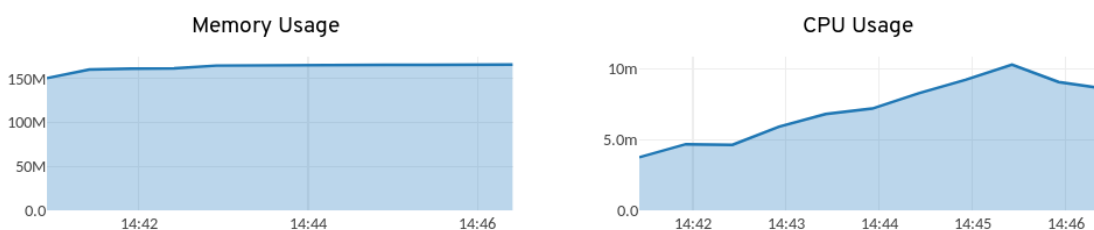
9. In the left hand side panel, expand **Workloads**.
10. Click **Pods** and then click **s2i-fuse75-spring-boot-camel-xml-xxxx**. The pod details for the application are displayed.

[s2i-fuse75-spring-boot-camel-xml-1](#) > Pod Details

P s2i-fuse75-spring-boot-camel-xml-1-66xfb

[Overview](#) [YAML](#) [Environment](#) [Logs](#) [Events](#) [Terminal](#)

Pod Overview




NAME
s2i-fuse75-spring-boot-camel-xml-1-66xfb

STATUS
Running

NAMESPACE
NS openshift

RESTART POLICY
Always Restart

11. To shut down the running pod,
 - a. On the Project Status page for the **openshift** project, click **s2i-fuse75-spring-boot-camel-xml-xxxx** application.
 - b. Click the **Overview** tab to view to the overview information page of the application.

- c. Click the  icon next to Desired Count. The Edit Count window is displayed.
- d. Use the down arrow to scale down to zero to stop the pod.

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-750020-redhat-00002/archetypes-catalog-2.2.0.fuse-750020-redhat-
00002-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-xml-archetype \
-DarchetypeVersion=2.2.0.fuse-750020-redhat-00002
```

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': : fuse75-spring-boot
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': org.example.fis: :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse75-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse75-spring-boot** for the **artifactId** value. Accept the defaults for the remaining fields.

3. If the above command exited with the BUILD SUCCESS status, you should now have a new Fuse on OpenShift project under the **fuse75-spring-boot** subdirectory.
4. You are now ready to build and deploy the **fuse75-spring-boot** project. Assuming you are still logged into OpenShift, change to the directory of the **fuse75-spring-boot** project, and then build and deploy the project, as follows.

```
cd fuse75-spring-boot
mvn fabric8:deploy -Popenshift
```



NOTE

For the full list of available Spring Boot archetypes, see [Spring Boot Archetype Catalog](#).

5.2. CREATING A SPRING BOOT 2 PROJECT USING MAVEN ARCHETYPE

This quickstart demonstrates how to create a Spring Boot 2 project using Maven archetypes.

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 2 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-sb2-750011-redhat-00006/archetypes-catalog-2.2.0.fuse-sb2-750011-
redhat-00006-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-xml-archetype \
-DarchetypeVersion=2.2.0.fuse-sb2-750011-redhat-00006
```

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': : fuse75-spring-boot
Define value for property 'version': : 1.0-SNAPSHOT: :
Define value for property 'package': : org.example.fis: :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse75-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse75-spring-boot** for the **artifactId** value. Accept the defaults for the remaining fields.

3. If the above command exited with the BUILD SUCCESS status, you should now have a new Fuse on OpenShift project under the **fuse75-spring-boot** subdirectory.
4. You are now ready to build and deploy the **fuse75-spring-boot** project. Assuming you are still logged into OpenShift, change to the directory of the **fuse75-spring-boot** project, and then build and deploy the project, as follows.

```
cd fuse75-spring-boot
mvn fabric8:deploy -Popenshift
```

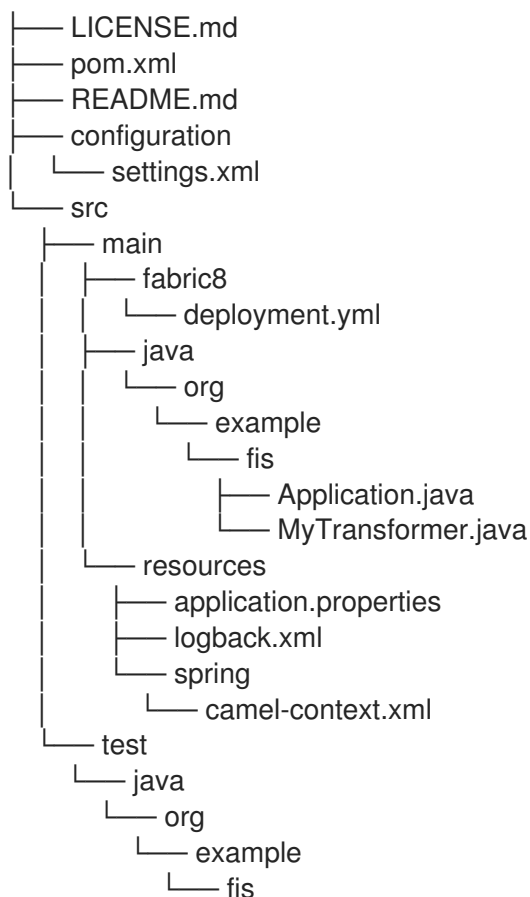


NOTE

For the full list of available Spring Boot 2 archetypes, see [Spring Boot 2 Archetype Catalog](#).

5.3. STRUCTURE OF THE CAMEL SPRING BOOT APPLICATION

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



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"})
```

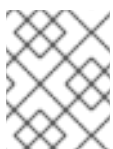
It must have a **main** method to run the Spring Boot application.

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.

5.4. 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-3scale-archetype | Demonstrates how to use Camel's REST DSL to expose a RESTful API and expose it to 3scale. |
| 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. |

| Name | Description |
|--|---|
| 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](#).

5.5. SPRING BOOT 2 ARCHETYPE CATALOG

The Spring Boot 2 Archetype catalog includes the following examples.

Table 5.2. Spring Boot 2 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-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-3scale-archetype | Demonstrates how to use Camel's REST DSL to expose a RESTful API and expose it to 3scale. |
| spring-boot-camel-rest-sql-archetype | Demonstrates how to use SQL via JDBC along with Camel's REST DSL to expose a RESTful API. |

| Name | Description |
|--|--|
| 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. |

NOTE

The following Spring Boot 2 Maven archetypes fail to build and deploy on to the OpenShift. See the [Release Notes](#) for more information.

- **spring-boot-camel-archetype**
- **spring-boot-camel-infinispan-archetype**
- **spring-boot-cxf-jaxrs-archetype**
- **spring-boot-cxf-jaxws-archetype**

To work around this issue, after generating a Maven project for one of these quickstarts, edit the project's Maven **pom.xml** file to add the following dependency:

```
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <version>2.4.1</version>
  <scope>test</scope>
</dependency>
```

5.6. BOM FILE FOR SPRING BOOT

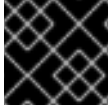
The purpose of a [Maven Bill of Materials \(BOM\)](#) file is to provide a curated set of Maven dependency versions that work well together, preventing you from having to define versions individually for every Maven artifact.

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.

5.6.1. 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>7.5.0.fuse-sb2-750029-redhat-00003</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>7.5.0.fuse-750029-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.

5.7. 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>7.5.0.fuse-750029-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. RUNNING APACHE CAMEL APPLICATION IN SPRING BOOT

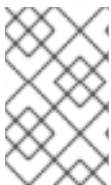
The Apache Camel Spring Boot component automatically configures Camel context for Spring Boot. Auto-configuration of the Camel context automatically detects the Camel routes available in the Spring context and registers the key Camel utilities such as producer template, consumer template, and the type converter as beans. The Apache Camel component includes a Spring Boot starter module that allows you to develop Spring Boot applications by using starters.

6.1. INTRODUCTION TO THE CAMEL SPRING BOOT COMPONENT

Every Camel Spring Boot application must use the **dependencyManagement** element in the project's **pom.xml** to specify the productized versions of the dependencies. These dependencies are defined in the Red Hat Fuse BOM and are supported for the specific version of Red Hat Fuse. You can omit the version number attribute for the additional starters so as not to override the versions from BOM. See [quickstart pom](#) for more information.

Example

```
<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

The **camel-spring-boot** jar contains with the **spring.factories** file which allows you to add that dependency to your classpath so Spring Boot will automatically configure Camel context.

6.2. INTRODUCTION TO THE CAMEL SPRING BOOT STARTER MODULE

Starters are the Apache Camel modules that are intended to be used in Spring Boot applications. There is a **camel-xxx-starter** module for each Camel component (with a few exceptions listed in the [Section 6.3, "List of the Camel components that do not have starter modules"](#) section).

Starters meet the following requirements:

- Allow auto-configuration of the component by using the native Spring Boot configuration system which is compatible with IDE tooling.
- Allow auto-configuration of data formats and languages.
- Manage transitive logging dependencies to integrate with the Spring Boot logging system.

- Include additional dependencies and align transitive dependencies to minimize the effort of creating a working Spring Boot application.

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



NOTE

For more details, see link: [Apache Camel Spring-Boot examples](#).

6.3. LIST OF THE CAMEL COMPONENTS THAT DO NOT HAVE STARTER MODULES

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

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

6.4. USING CAMEL SPRING BOOT STARTER

Apache Camel provides a starter module that allows you to quickly get started developing Spring Boot applications.

Procedure

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

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

2. Add the 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");
    }
}

```

3. Optional. To keep the main thread blocked so that Camel stays up, do one of the following.
 - a. Include the **spring-boot-starter-web** dependency,
 - b. Or add **camel.springboot.main-run-controller=true** to your **application.properties** or **application.yml** file.
You can customize the Camel application in the **application.properties** or **application.yml** file with **camel.springboot.* properties**.
4. Optional. To refer to a custom bean by using the bean's ID name, configure the options in the **src/main/resources/application.properties** (or the **application.yml**) file. The following example shows how the xslt component refers to a custom bean by using the bean ID.
 - a. Refer to a custom bean by the id **myExtensionFactory**.

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

- b. Then create the custom bean using Spring Boot `@Bean` annotation.

```

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

```

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

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

6.5. ABOUT CAMEL CONTEXT AUTO-CONFIGURATION FOR SPRING BOOT

Camel Spring Boot 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. You can access the **camelContext** as shown below.

Example

```
@Configuration
```

```
public class MyAppConfig {  
  
    @Autowired  
    CamelContext camelContext;  
  
    @Bean  
    MyService myService() {  
        return new DefaultMyService(camelContext);  
    }  
  
}
```

6.6. AUTO-DETECTING CAMEL ROUTES IN SPRING BOOT APPLICATIONS

Camel auto-configuration collects all the **RouteBuilder** instances from the Spring context and automatically injects them into the **CamelContext**. This simplifies the process of creating a new Camel route with the Spring Boot starter. You can create the routes as follows:

Example

Add 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");  
    }  
  
}
```

Or create a new route **RouteBuilder** bean in your **@Configuration** class.

```
@Configuration  
public class MyRouterConfiguration {  
  
    @Bean  
    RoutesBuilder myRouter() {  
        return new RouteBuilder() {  
  
            @Override  
            public void configure() throws Exception {  
                from("jms:invoices").to("file:/invoices");  
            }  
  
        };  
    }  
  
}
```


6.7. CONFIGURING CAMEL PROPERTIES FOR CAMEL SPRING BOOT AUTO-CONFIGURATION

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

Procedure

1. Define the properties either in the **application.properties** file:

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

Or set the Camel properties as the system properties, for example:

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

2. Use the configured properties as placeholders in Camel route as follows.

```
@Component
public class MyRouter extends RouteBuilder {

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

6.8. CONFIGURING CUSTOM CAMEL CONTEXT

To perform operations on the **CamelContext** bean created by Camel Spring Boot auto-configuration, register a **CamelContextConfiguration** instance in your Spring context.

Procedure

- Register an instance of **CamelContextConfiguration** in the 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
            }
        };
    }
}
```

The **CamelContextConfiguration** and **beforeApplicationStart(CamelContext)** methods are called before the Spring context is started, so the **CamelContext** instance that is 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.

6.9. DISABLING JMX IN THE AUTO-CONFIGURED CAMELCONTEXT

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

Procedure

- Add the following property to your **application.properties** file and set it to **false**:

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

6.10. INJECTING AUTO-CONFIGURED CONSUMER AND PRODUCER TEMPLATES INTO SPRING-MANAGED BEANS

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

Example

```
@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 these values by setting the following Spring properties to the desired cache size, for example:

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

6.11. ABOUT THE AUTO-CONFIGURED TYPECONVERTER IN THE SPRING CONTEXT

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

Example

```

@Component
public class InvoiceProcessor {

    @Autowired
    private TypeConverter typeConverter;

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

```

6.12. SPRING TYPE CONVERSION API BRIDGE

Spring consist of a powerful [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. This 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:

Example

```

@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**.

6.13. DISABLING TYPE CONVERSIONS FEATURES

To disable the Camel Spring Boot type conversion features, set the **camel.springboot.typeConversion** property to **false**. When this property is set to **false**, the auto-configuration does not register a type converter instance and does not enable the delegation of type conversion to the Spring Boot type conversion API.

Procedure

- To disable the type conversion features of Camel Spring Boot component, set the **camel.springboot.typeConversion** property to **false** as shown below:

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

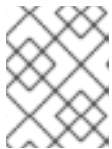
6.14. ADDING XML ROUTES TO THE CLASSPATH FOR AUTO-CONFIGURATION

By default, the Camel Spring Boot component auto-detects and includes the Camel XML routes that are in the classpath in the **camel** directory. You can configure the directory name or disable this feature using the configuration option.

Procedure

- Configure the Camel Spring Boot XML routes in the classpath as follows.

```
// 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 define the Camel XML route elements and not **CamelContext** elements, for example:

```
<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>
```

Using Spring XML files

To use Spring XML files with the `<camelContext>`, you can configure a Camel context in the Spring XML file or in the **application.properties** file. To set the name of the Camel context and turn on the stream caching, add the following in the **application.properties** file:

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

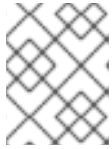
6.15. ADDING XML REST-DSL ROUTES FOR AUTO-CONFIGURATION

The Camel Spring Boot component auto-detects and embeds the Camel Rest-DSL XML routes that are added in the classpath under the **camel-rest** directory. You can configure the directory name or disable this feature using the configuration option.

Procedure

- Configure the Camel Spring Boot Rest-DSL XML routes in the classpath as follows.

```
// 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 define the Camel XML REST elements and not **CamelContext** elements, for example:

```
<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:getPersonId"/>
    </get>
    <put uri="/persons/{personId}">
      <to uri="direct:putPersonId"/>
    </put>
    <delete uri="/persons/{personId}">
      <to uri="direct:deletePersonId"/>
    </delete>
  </rest>
</rests>
```

6.16. TESTING WITH CAMEL SPRING BOOT

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

When you have multiple Camel routes in different RouteBuilder classes, the Camel Spring Boot component automatically embeds all these routes when running the application. 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.

Procedure

1. Specify the **include** or **exclude** 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 a double asterisk, which matches with any leading package name. **/Foo*** means the class name must start with Foo, for example, FooRoute.

2. Run the test using the following maven command:

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

Additional Resources

- [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 OPENSIFT

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 --branch spring-boot-camel-xa-1.0.0.fuse-750020-redhat-00005
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.

- Install **postgresql** using username as **theuser** and password as **Thepassword1!**.

```
oc new-app --param=POSTGRESQL_USER=theuser --
param=POSTGRESQL_PASSWORD='Thepassword1!' --
env=POSTGRESQL_MAX_PREPARED_TRANSACTIONS=100 --template=postgresql-
persistent
```

- Install the **A-MQ** broker using username as **theuser** and password as **Thepassword1!**.

```
oc new-app --param=MQ_USERNAME=theuser --
param=MQ_PASSWORD='Thepassword1!' --template=amq63-persistent
```

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

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

7. Build and deploy your quickstart.

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

8. 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 -).

9. 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 quickstart demonstrates how to connect a Spring Boot application to AMQ Online and use JMS messaging between two Camel routes using Fuse on OpenShift.

Prerequisites

- Ensure that AMQ Online is deployed and running. To install AMQ Online on OpenShift see [Installing and configuring AMQ Online on OpenShift](#).
- 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. Log in to the OpenShift server as a developer.

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

2. Log in to the **openshift** project.

```
oc project openshift
```

3. 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-750020-redhat-00002/archetypes-catalog-2.2.0.fuse-750020-redhat-
00002-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-amq-archetype \
-DarchetypeVersion=2.2.0.fuse-750020-redhat-00002
```

4. 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': : fuse75-spring-boot-camel-amq
Define value for property 'version': : 1.0-SNAPSHOT: :
Define value for property 'package': : org.example.fis: :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse75-spring-boot-camel-amq
```

```
version: 1.0-SNAPSHOT
package: org.example.fis
Y: :Y
```

When prompted, enter **org.example.fis** for the **groupid** value and **fuse75-spring-boot-camel-amq** for the **artifactId** value. Accept the defaults for the remaining fields.

5. Navigate to the quickstart directory **fuse75-spring-boot-camel-amq**.

```
cd fuse75-spring-boot-camel-amq
```

6. Run the following commands to apply configuration files to AMQ Online. These configuration files create the AMQ Online user and the queue, both with the admin privileges.

```
oc login -u system:admin
oc apply -f src/main/resources/k8s
```

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

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

8. To verify that the quickstart is running successfully:

- a. Navigate to the OpenShift web console in your browser (https://OPENSIFT_IP_ADDR, replace OPENSIFT_IP_ADDR with the IP address of the cluster) and log in to the console with your credentials (for example, with username developer and password, developer).
- b. In the left hand side panel, expand **Home** and then click **Status** to view the Project Status page for **openshift** project.
- c. Click **fuse75-spring-boot-camel-amq** to view the Overview information page for the quickstart.
- d. In the left hand side panel, expand **Workloads**.
- e. Click **Pods** and then click **fuse75-spring-boot-camel-amq-xxxxx**. The pod details for the quickstart are displayed.
- f. Click **Logs** to view the logs for the application.
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
```

9. To view the routes on the web interface, click **Open Java Console** and check the messages in the AMQ queue.

CHAPTER 9. INTEGRATING SPRING BOOT WITH KUBERNETES

The Spring Cloud Kubernetes plugin 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 plugin

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>

```

- 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

- 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-750020-redhat-00002/archetypes-catalog-2.2.0.fuse-750020-redhat-
00002-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-config-archetype \
-DarchetypeVersion=2.2.0.fuse-750020-redhat-00002

```

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': : fuse75-configmap

```

```

Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': org.example.fis: :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse75-configmap
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y

```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse75-configmap** for the **artifactId** value. Accept the defaults for the remaining fields.

- 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 **openshift** project, enter the following commands:

```

oc login -u developer -p developer
oc project openshift

```

- At the command line, change to the directory of the new **fuse75-configmap** project and create the Secret object for this application.

```

cd fuse75-configmap
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](#).

- Build and deploy the quickstart application. From the top level of the **fuse75-configmap** project, enter:

```

mvn fabric8:deploy -Popenshift

```

- View the application log as follows.
 - Navigate to the OpenShift web console in your browser (https://OPENSHIFT_IP_ADDR, replace **OPENSHIFT_IP_ADDR** with the IP address of the cluster) and log in to the console with your credentials (for example, with username **developer** and password, **developer**).
 - In the left hand side panel, expand **Home**. Click **Status** to view the **Project Status** page. All the existing applications in the selected namespace (for example, openshift) are displayed.
 - Click **fuse75-configmap** to view the **Overview** information page for the quickstart.
 - In the left hand side panel, expand **Workloads**.
 - Click **Pods** and then click **fuse75-configmap-xxxx**. The pod details for the application are displayed.
 - Click on the **Logs** tab to view the application logs.

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 `fuse75-configmap` application using a ConfigMap object, you must give the `fuse75-configmap` application permission to view data from the OpenShift ApiServer. Enter the following command to give the `view` permission to the `fuse75-configmap` application's service account:

```
oc policy add-role-to-user view system:serviceaccount:openshift: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:

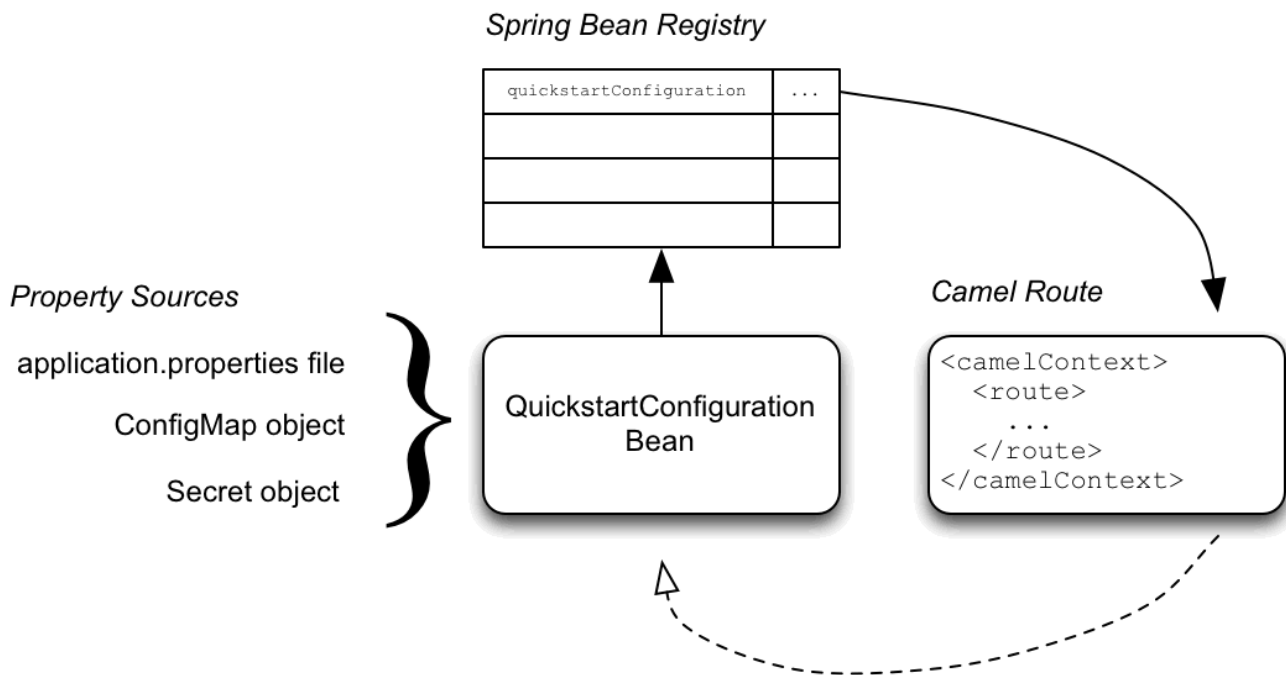
```
16:25:24.121 [Camel (camel) thread #0 - timer://order] INFO generate-order-route -
Generating message message-9, sending to the recipient list
16:25:24.124 [Camel (camel) thread #0 - timer://order] INFO target-route-queue - ---->
message-9 pushed to an async queue (simulation)
16:25:24.125 [Camel (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 configuration 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 **fuse75-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: ❶
  name: camel-config
type: Opaque
data:
  # The username is 'myuser'
  quickstart.queue-username: bXI1c2VyCg== ❷
  quickstart.queue-password: MWYyZDFIMmU2N2Rm ❸

```

- ❶ `metadata.name`: Identifies the Secret. Other parts of the OpenShift system use this identifier to reference the Secret.
- ❷ `quickstart.queue-username`: Is meant to be injected into the `queueUsername` property of the `quickstartConfiguration` bean. The value *must* be base64 encoded.
- ❸ `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: ❶
      - name: "camel-config"
        secret:
          # The secret must be created before deploying this application
          secretName: "camel-config"
    containers:
      -
        volumeMounts: ❷
        - 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: ❶
  # Must match the 'spring.application.name' property of the application
  name: camel-config
data:
  application.properties: | ❷
    # Override the configuration properties here
    quickstart.recipients=direct:async-queue,direct:file,direct:mail ❸
```

- ❶ **metadata.name**: Identifies the ConfigMap. Other parts of the OpenShift system use this identifier to reference the ConfigMap.
- ❷ **data.application.properties**: This section lists property settings that can override settings from the original **application.properties** file that was deployed with the application.
- ❸ **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 application.yaml ConfigMap property

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 application.properties ConfigMap property

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 PostgreSQL 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: MWYyZDFIMmU2N2Rm
```

PostgreSQL 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-750020-redhat-00002/archetypes-catalog-2.2.0.fuse-750020-redhat-
00002-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=karaf-camel-log-archetype \
-DarchetypeVersion=2.2.0.fuse-750020-redhat-00002
```

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': : fuse75-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: fuse75-karaf-camel-log
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse75-karaf-camel-log** for the **artifactId** value. Accept the defaults for the remaining fields.

4. If the above command exited with the BUILD SUCCESS status, you should now have a new Fuse on OpenShift project under the **fuse75-karaf-camel-log** subdirectory.
5. You are now ready to build and deploy the **fuse75-karaf-camel-log** project. Assuming you are still logged into OpenShift, change to the directory of the **fuse75-karaf-camel-log** project, and then build and deploy the project, as follows.

```
cd fuse75-karaf-camel-log
mvn fabric8:deploy -Popenshift
```

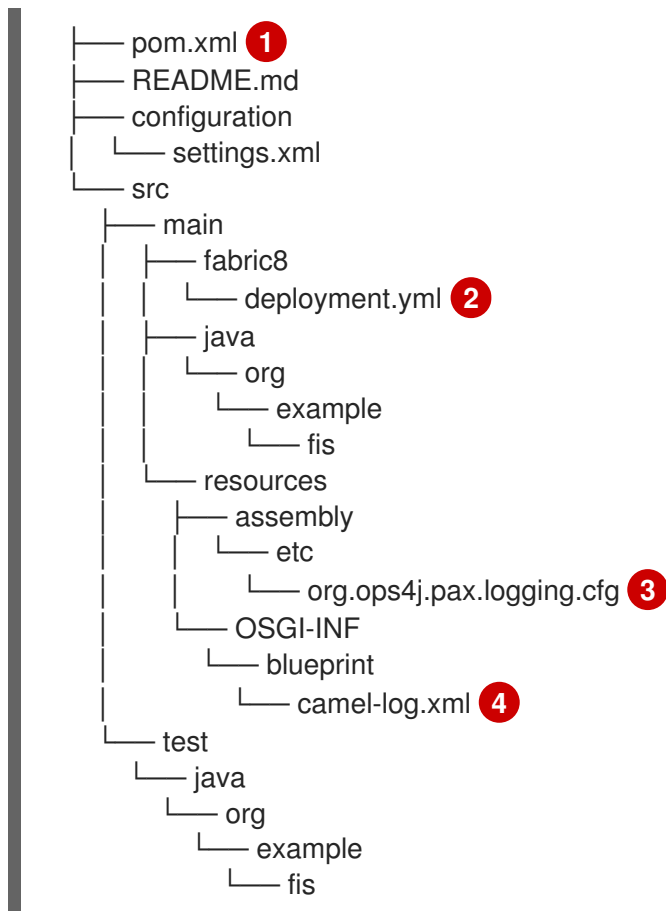


NOTE

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:



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 health 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
     * @return true if altered
     */
    boolean replaceIn(StringBuilder value);

```



```

/**
 * Replaces all the occurrences of variables within the given dictionary
 *
 * @param dictionary the dictionary to replace in
 * @return 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
 * @return 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. |
| <code>`sys</code> | sys:java.version | look up the property from Java JVM system properties. |
| <code>`service</code> | 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 | <code>[\$[</code> | The prefix for the placeholder |
| fabric8.placeholder.suffix | <code>]]</code> | The suffix for the placeholder |
| fabric8.k8s.secrets.path | null | A comma delimited list of paths where secrets are mapped |

| Name | Default | Description |
|--|---------|---|
| 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"/>

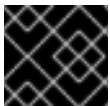
  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="userName" value="${amq.usr}"/>
    <property name="password" value="${amq.pwd}"/>
    <property name="brokerURL" value="${amq.url}"/>
  </bean>
</blueprint>

```

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 |
| 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 | <p>Define additional conditions for the ConfigAdmin bridge to select a ConfigMap. The supported syntax is:</p> <ul style="list-style-type: none"> • Conditions on different labels are separated by "," and are intended in AND between each other. • Inside a label, semicolons (;) are considered as OR and can be used as separators for conditions on the label value. <p>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".</p> |



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

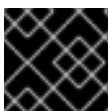
The fabric8 evaluator supports chained evaluators, such as **`\${env+service:MY_ENV_VAR}**. You need to resolve **MY_ENV_VAR** variable against environment variables. The result is then resolved using service function. For example,

```
<?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 your 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 you are 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 health 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. View the installed Fuse on OpenShift templates.

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

3. Enter the following command to create the resources required for running the **Red Hat Fuse 7.5 Camel CDI with EAP** quickstart. It creates a deployment config and build config for the quickstart. The information about the quickstart and the resources created is displayed on the terminal.

```
oc new-app s2i-fuse75-eap-camel-cdi

--> Creating resources ...
service "s2i-fuse75-eap-camel-cdi" created
service "s2i-fuse75-eap-camel-cdi-ping" created
route.route.openshift.io "s2i-fuse75-eap-camel-cdi" created
imagestream.image.openshift.io "s2i-fuse75-eap-camel-cdi" created
buildconfig.build.openshift.io "s2i-fuse75-eap-camel-cdi" created
deploymentconfig.apps.openshift.io "s2i-fuse75-eap-camel-cdi" created
--> Success
Access your application via route 's2i-fuse75-eap-camel-cdi-OPENSHIFT_IP_ADDR'
Build scheduled, use 'oc logs -f bc/s2i-fuse75-eap-camel-cdi' to track its progress.
Run 'oc status' to view your app.
```

4. Navigate to the OpenShift web console in your browser (https://OPENSIFT_IP_ADDR, replace **OPENSIFT_IP_ADDR** with the IP address of the cluster) and log in to the console with your credentials (for example, with username **developer** and password, **developer**).
5. In the left hand side panel, expand **Home**. Click **Status** to view the **Project Status** page. All the existing applications in the selected namespace (for example, openshift) are displayed.
6. Click **s2i-fuse75-eap-camel-cdi** to view the **Overview** information page for the quickstart.

The screenshot shows the OpenShift Project Status page. On the left, the 'Project Status' section is visible, with 'Resources' selected. The application 's2i-fuse75-eap-camel-cdi' is listed with '1 of 1 pods'. On the right, the 'Overview' page for 's2i-fuse75-eap-camel-cdi' is shown. It includes a table with the following data:

| DESIRED COUNT | UP-TO-DATE COUNT | MATCHING PODS | |
|---------------|------------------|---------------|------------------------------|
| 1 pod | 1 pod | 1 pod | 0 available 1 unavailable |

Below the table, there is a list of resources with the following columns: NAME and LATEST VERSION.

| NAME | LATEST VERSION |
|--------------------------|----------------|
| s2i-fuse75-eap-camel-cdi | 1 |

7. Click the **Resources** tab and then click the link displayed in the Routes section to access the application.



DC s2i-fuse75-eap-camel-cdi

Actions ▾

Overview

Resources

Builds

BC s2i-fuse75-eap-camel-cdi

Start Build

✔ Build #1 is complete (less than a minute ago)

[View Logs](#)

Services

S s2i-fuse75-eap-camel-cdi

Service port: TCP/8080 → Pod Port: 8080

S s2i-fuse75-eap-camel-cdi-ping

Service port: ping → Pod Port: 8888

Routes

RT s2i-fuse75-eap-camel-cdi

Location:

<http://s2i-fuse75-eap-camel-cdi-openshift.apps.test-mycluster01.devcluster.openshift.com> ↗

The link has the form http://s2i-fuse75-eap-camel-cdi-OPENSHIFT_IP_ADDR. 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-fuse75-eap-camel-cdi-openshift.apps.cluster-name.openshift.com/?name=jdoe>, in your browser you see the response:

```
Hello jdoe from 172.17.0.3
```

- Click **View Logs** to view the logs for the application.
- To shut down the running pod,

```
oc delete pod $(oc get pods -l deployment=s2i-fuse75-eap-camel-cdi -o name)
```

- a. Click the **Overview** tab to return to the overview information page of the application.



- b. Click the  icon next to Desired Count. The **Edit Count** window is displayed.

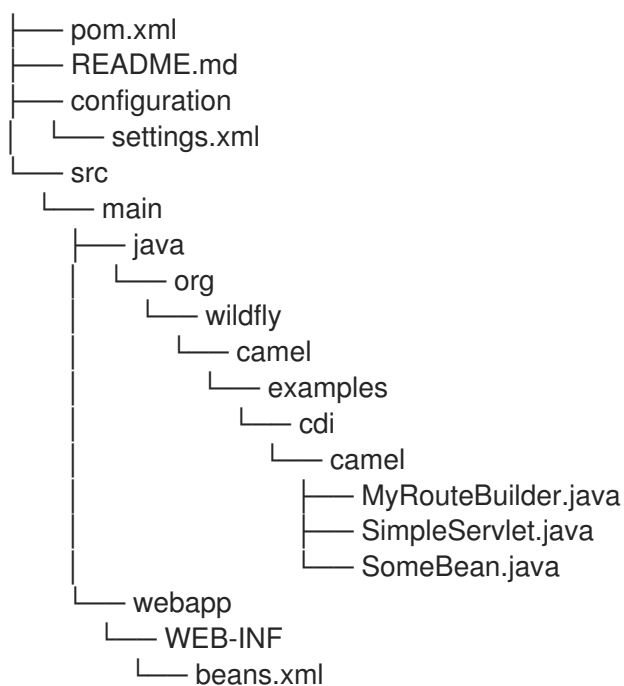
- c. Use the down arrow to scale down to zero to stop the pod.

11.2. STRUCTURE OF THE JBOSS EAP APPLICATION

You can find the source code for the Red Hat Fuse 7.5 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:



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 |
|------|-------------|
| | |

| Name | Description |
|--|--|
| JBoss Fuse 7.5 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 7.5 Camel CDI with EAP (eap-camel-cdi-template) | Demonstrates using the camel-cdi component to integrate CDI beans with Camel routes. |
| Red Hat Fuse 7.5 CXF JAX-RS with EAP (eap-camel-cxf-jaxrs-template) | Demonstrates using the camel-cxf component to produce and consume JAX-RS REST services. |
| Red Hat Fuse 7.5 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 7.5 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 a client to the database. |

CHAPTER 12. USING PERSISTENT STORAGE IN FUSE ON OPENSIFT

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 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 manage 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 |
|--------------------------|--|-----------------|
| UPDATE | | |
| fuse7-console | 172.30.1.1:5000/openshift/fuse7-console | |
| 1.0,1.1,1.2,1.3,1.4,1.5 | About an hour ago | |
| fuse7-eap-openshift | 172.30.1.1:5000/openshift/fuse7-eap-openshift | |
| 1.0,1.1,1.2,,1.3,1.4,1.5 | About an hour ago | |
| fuse7-java-openshift | 172.30.1.1:5000/openshift/fuse7-java-openshift | |
| 1.0,1.1,1.2,1.3,1.4,1.5 | About an hour ago | |
| fuse7-karaf-openshift | 172.30.1.1:5000/openshift/fuse7-karaf-openshift | |
| 1.0,1.1,1.2,1.3,1.4,1.5 | About an hour ago... | |
| fuse-apicurito-generator | 172.30.1.1:5000/openshift/fuse-apicurito-generator | |
| 1.2,1.3,1.4,1.5 | About an hour ago... | |
| apicurito-ui | 172.30.1.1:5000/openshift/apicurito-ui | 1.2,1.3,1.4,1.5 |
| About an hour ago... | | |

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

```
oc import-image -n openshift fuse7-java-openshift:1.5
oc import-image -n openshift fuse7-karaf-openshift:1.5
oc import-image -n openshift fuse7-eap-openshift:1.5
oc import-image -n openshift fuse7-console:1.5
oc import-image -n openshift apicurito-ui:1.5
oc import-image -n openshift fuse-apicurito-generator:1.5
```



NOTE

The version tags in the image stream have the form **1.5-<BUILDNUMBER>**. When you specify the tag as **1.5**, you will get the latest build in the **1.5** 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 OPENSIFT 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-750056-redhat-00006
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>7.5.0.fuse-750029-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>7.5.0.fuse-750029-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>7.5.0.fuse-750029-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 | 7.5.0.fuse-750029-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 PLUGIN

Spring Boot Maven plugin 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.1. SPRING BOOT MAVEN PLUGIN GOALS

The Spring Boot Maven plugin includes the following goals:

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

A.2. USING SPRING BOOT MAVEN PLUGIN

You can find general instructions on how to use the Spring Boot Plugin at: <https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/#using>. The following examples 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 <https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/link>.

A.2.1. Using Spring Boot Maven plugin for Spring Boot 2

The following example illustrates the usage of the **spring-boot-maven-plugin** for Spring Boot 2.

Example

```
<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>7.5.0.fuse-sb2-750029-redhat-00003</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.2.2. Using Spring Boot Maven plugin for Spring Boot 1

The following example illustrates the usage of the **spring-boot-maven-plugin** for Spring Boot 1.

Example

```

<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>7.5.0.fuse-750029-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>
</project>

```

```
        </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.

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 archive) 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 PLUGIN

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. BUILDING IMAGES

The **fabric8:build** goal is used 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.2. KUBERNETES AND OPENSIFT 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**.

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.3. 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.

Procedure

- To connect the Fabric8 Maven plugin to pre- and post-integration phases, add the following to **settings.xml** file.

```
<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.4. UNDERSTANDING FABRIC8 MAVEN PLUGIN 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.4.1. 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 |
|-------------------------------------|--|
| <code>fabric8:build</code> | Build images. Note that Fuse on OpenShift supports only the OpenShift s2i build strategy, not the docker build strategy. |
| <code>fabric8:resource</code> | Create Kubernetes or OpenShift resource descriptors |
| <code>fabric8:apply</code> | Apply resources to a running cluster |
| <code>fabric8:resource-apply</code> | Run fabric8:resource → fabric8:apply |

Table C.2. Development Goals

| Goal | Description |
|-------------------------------|---|
| <code>fabric8:run</code> | Run a complete development workflow cycle fabric8:resource → fabric8:build → fabric8:apply in the foreground. |
| <code>fabric8:deploy</code> | Deploy resources descriptors to a cluster after creating them and building the app. Same as fabric8:run except that it runs in the background. |
| <code>fabric8:undeploy</code> | Undeploy and remove resources descriptors from a cluster. |
| <code>fabric8:start</code> | Start the application which has been deployed previously |
| <code>fabric8:stop</code> | Stop the application which has been deployed previously |
| <code>fabric8:log</code> | Show the logs of the running application |
| <code>fabric8:debug</code> | Enable remote debugging |
| <code>fabric8:watch</code> | Monitor the project workspace for changes and automatically trigger redeployment of application. |

C.4.2. Setting environmental variable

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

Example

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

```

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

```

C.4.3. 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.5. GENERATORS

The Fabric8 Maven plugin provides *generator* components, which have the capability to build images automatically for specific kinds of application. Following generator types are supported in Fuse on OpenShift:

- [Section C.5.3, "Spring Boot"](#)
- [Section C.5.4, "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.5.1. Zero configuration

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

C.5.2. 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.5.2.1. Default values for istag mode

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

```
fuse7/fuse-eap-openshift:1.5
fuse7/fuse-java-openshift:1.5
fuse7/fuse-karaf-openshift:1.5
```



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.5.2.2. 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 plugin uses the following default image specifiers to select the Fuse images (formatted as `<image-namespace>/<image-name>:<tag>`):

```
fuse7/fuse-eap-openshift:1.5
fuse7/fuse-java-openshift:1.5
fuse7/fuse-karaf-openshift:1.5
```

C.5.2.3. 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:

Example

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

C.5.2.4. 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:

Example

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

C.5.2.5. Specifying the Generator mode using 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.

Example

```
//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.5.3. Spring Boot

The Spring Boot generator gets activated when it finds a **spring-boot-maven-plugin** plugin in the **pom.xml** file. The generated container port is read from the **server.port** property in the **application.properties** file, 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 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. | |
| 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.5.4. Karaf

The Karaf generator gets activated when it finds a **karaf-maven-plugin** plugin 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 PLUGIN

You can use `fabric8-camel-maven` plugin to validate all your Camel endpoints from the source code. This allows you to ensure that the endpoints are valid before you run your Camel applications or unit tests.

D.1. FABRIC8 CAMEL MAVEN PLUGIN GOALS

For validating Camel endpoints in the source code use:

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

D.2. ADDING THE FABRIC8-CAMEL-MAVEN PLUGIN TO YOUR PROJECT

You can add the `fabric8-camel-maven` plugin to your project by adding it to your project's `pom.xml` file.

Procedure

1. 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>.

2. Then you can run the `validate` goal from the command line or from your Java editor such as IDEA or Eclipse.

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

Running the plugin automatically

You can also enable the Plugin to run automatically as a part of the build to catch the errors. In the following example, the phase determines when the Plugin runs. In the example, the phase is **process-classes** which runs after the compilation of the main source code.

Example

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

```

</execution>
</executions>
</plugin>

```

Validating the test 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.

Example

```

<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.

Procedure

- To run the goal on the **camel-example-cdi** plugin from Apache Camel, run the following commands:

```

$cd camel-example-cdi
$mvn io.fabric8.forge:fabric8-camel-maven-plugin:2.3.80:validate

```

This displays the following output:

```

[INFO] -----
[INFO] Building Camel :: Example :: CDI 2.16.2
[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. Following example shows how to validate and if required, correct the camel endpoints.

Example

Let us assume that you made a typo in one of the Camel endpoint URIs in the source code, such as:

1. The correct Camel endpoint URI is as follows.

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

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

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

3. Run the validate goal again.

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

As shown above the error in the camel endpoint URI is displayed.

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.

Table D.1. Fabric8 Camel Maven plugin options

| 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. |

| Parameter | Default Value | Description |
|-------------------------|---------------|--|
| 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. |
| 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.

Procedure

- 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. CUSTOMIZING JVM ENVIRONMENT VARIABLES

You can use JVM environment variables to set all the options for the Fuse on OpenShift images.

E.1. USING S2I JAVA BUILDER IMAGE WITH OPENJDK 8

Using the S2I Java builder image you can run results directly without using any other application server. This S2I image is suitable for microservices with a flat classpath (including **fat jars**).

You can configure Java options when using the Fuse on OpenShift images. 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. USING S2I KARAF BUILDER IMAGE WITH OPENJDK 8

The S2I Karaf builder image can be used with OpenShift's Source To Image workflow to build Karaf4 custom assembly based maven projects.

Procedure

- Use following command to use S2I workflow.

```
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 Maven 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. BUILD TIME ENVIRONMENT VARIABLES

Following are the environment variables that are used to influence the behaviour of S2I Java and Karaf builder images during the build time.

- **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, removes 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.4. RUN TIME ENVIRONMENT VARIABLES

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 manually restricts 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 use 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.5. 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_OPENSIFT**: 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

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.1. 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.2. DEFAULT BEHAVIOUR OF FUSE ON OPENSIFT 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 of the default heap size instead of the one-half.

F.3. CUSTOM TUNING OF FUSE ON OPENSIFT IMAGES

The script sets the `CONTAINER_MAX_MEMORY` and `CONTAINER_CORE_LIMIT` environment variables, which are 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.

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.4. 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