



Red Hat Fuse 7.9

Fuse on OpenShift Guide

Install and manage Red Hat Fuse on OpenShift, develop and deploy Fuse applications on OpenShift

Red Hat Fuse 7.9 Fuse on OpenShift Guide

Install and manage Red Hat Fuse on OpenShift, develop and deploy Fuse applications 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

MAKING OPEN SOURCE MORE INCLUSIVE	7
CHAPTER 1. BEFORE YOU BEGIN	8
1.1. COMPARISON: FUSE STANDALONE AND FUSE ON OPENSIFT	8
CHAPTER 2. GETTING STARTED FOR ADMINISTRATORS	10
2.1. AUTHENTICATING WITH REGISTRY.REDHAT.IO FOR CONTAINER IMAGES	10
2.2. INSTALLING FUSE IMAGESTREAMS AND TEMPLATES ON THE OPENSIFT 4.X SERVER	11
2.3. INSTALLING API DESIGNER ON OPENSIFT 4.X	14
2.3.1. Adding API Designer as a service to an OpenShift 4.x project	15
2.3.2. Upgrading the API Designer on OpenShift 4.x	16
2.3.3. Metering labels for API Designer	16
2.3.4. Considerations for installing API Designer in a restricted environment	17
2.4. SETTING UP THE FUSE CONSOLE ON OPENSIFT 4.X	18
2.4.1. Installing and deploying the Fuse Console on OpenShift 4.x by using the OperatorHub	18
2.4.2. Installing and deploying the Fuse Console on OpenShift 4.x by using the command line	20
2.4.2.1. Generating a certificate to secure the Fuse Console on OpenShift 4.x	22
2.4.3. Role-based access control for the Fuse Console on OpenShift 4.x	24
2.4.3.1. Determining access roles for the Fuse Console on OpenShift 4.x	24
2.4.3.2. Customizing role-based access to the Fuse Console on OpenShift 4.x	25
2.4.3.3. Disabling role-based access control for the Fuse Console on OpenShift 4.x	26
2.4.4. Upgrading the Fuse Console on OpenShift 4.x	27
2.5. CONFIGURING PROMETHEUS TO MONITOR FUSE APPLICATIONS ON OPENSIFT	28
2.5.1. About Prometheus	28
2.5.1.1. Prometheus queries	28
2.5.1.2. Options for displaying Prometheus data	29
2.5.2. Setting up Prometheus	29
2.5.3. OpenShift environment variables	31
2.5.4. Controlling the metrics that Prometheus monitors and collects	32
2.6. USING METERING FOR FUSE ON OPENSIFT	33
2.6.1. Metering resources	33
2.6.2. Metering labels for Fuse on OpenShift	33
2.7. MONITORING FUSE ON OPENSIFT WITH CUSTOM GRAFANA DASHBOARDS	34
2.8. INSTALLING FUSE IMAGESTREAMS AND TEMPLATES ON THE OPENSIFT 3.X SERVER	38
2.8.1. Setting up the Fuse Console on OpenShift 3.11	39
2.8.1.1. Deploying the Fuse Console on OpenShift 3.11	40
2.8.1.2. Monitoring a single Fuse pod from the Fuse Console on OpenShift 3.11	42
CHAPTER 3. INSTALLING FUSE ON OPENSIFT IN A RESTRICTED ENVIRONMENT	44
3.1. SETTING UP INTERNAL DOCKER REGISTRY	44
3.2. CONFIGURING INTERNAL REGISTRY SECRETS	45
3.3. INSTALLING FUSE ON OPENSIFT IMAGES IN A RESTRICTED ENVIRONMENT	45
3.4. USING AN INTERNAL MAVEN REPOSITORY	47
3.4.1. Running a Spring Boot application with MAVEN_MIRROR_URL	47
3.4.2. Running a Spring Boot application with OpenShift Maven plugin	47
CHAPTER 4. INSTALLING FUSE ON OPENSIFT AS A NON-ADMIN USER	49
4.1. INSTALLING FUSE ON OPENSIFT IMAGES AND TEMPLATES AS A NON-ADMIN USER	49
CHAPTER 5. GETTING STARTED FOR DEVELOPERS	52
5.1. PREPARING DEVELOPMENT ENVIRONMENT	52
5.1.1. Installing Container Development Kit (CDK) on your local machine	52

5.1.2. Getting remote access to an existing OpenShift server	53
5.1.3. Installing Client-Side tools	53
5.1.4. Configuring Maven repositories	54
5.2. CREATING AND DEPLOYING APPLICATIONS ON FUSE ON OPENSIFT	54
5.2.1. Creating and deploying an application using the S2I binary workflow	54
5.2.2. Undeploying and redeploying the project	58
5.2.3. Creating and deploying an application using the S2I source workflow	58
CHAPTER 6. DEVELOPING AN APPLICATION FOR THE SPRING BOOT IMAGE	62
6.1. CREATING A SPRING BOOT 2 PROJECT USING MAVEN ARCHETYPE	62
6.2. STRUCTURE OF THE CAMEL SPRING BOOT APPLICATION	63
6.3. SPRING BOOT 2 ARCHETYPE CATALOG	65
6.4. BOM FILE FOR SPRING BOOT	66
6.5. INCORPORATE THE BOM FILE	67
6.6. SPRING BOOT MAVEN PLUGIN	68
CHAPTER 7. RUNNING APACHE CAMEL APPLICATION IN SPRING BOOT	69
7.1. INTRODUCTION TO THE CAMEL SPRING BOOT COMPONENT	69
7.2. INTRODUCTION TO THE CAMEL SPRING BOOT STARTER MODULE	69
7.3. LIST OF THE CAMEL COMPONENTS THAT DO NOT HAVE STARTER MODULES	70
7.4. USING CAMEL SPRING BOOT STARTER	70
7.5. ABOUT CAMEL CONTEXT AUTO-CONFIGURATION FOR SPRING BOOT	71
7.6. AUTO-DETECTING CAMEL ROUTES IN SPRING BOOT APPLICATIONS	72
7.7. CONFIGURING CAMEL PROPERTIES FOR CAMEL SPRING BOOT AUTO-CONFIGURATION	73
7.8. CONFIGURING CUSTOM CAMEL CONTEXT	73
7.9. DISABLING JMX IN THE AUTO-CONFIGURED CAMELCONTEXT	74
7.10. INJECTING AUTO-CONFIGURED CONSUMER AND PRODUCER TEMPLATES INTO SPRING-MANAGED BEANS	74
7.11. ABOUT THE AUTO-CONFIGURED TYPECONVERTER IN THE SPRING CONTEXT	74
7.12. SPRING TYPE CONVERSION API BRIDGE	75
7.13. DISABLING TYPE CONVERSIONS FEATURES	75
7.14. ADDING XML ROUTES TO THE CLASSPATH FOR AUTO-CONFIGURATION	76
7.15. ADDING XML REST-DSL ROUTES FOR AUTO-CONFIGURATION	76
7.16. TESTING WITH CAMEL SPRING BOOT	77
CHAPTER 8. RUNNING SOAP TO REST BRIDGE QUICKSTART FOR SPRING BOOT 2 ON FUSE ON OPENSIFT	79
CHAPTER 9. RUNNING A CAMEL SERVICE ON SPRING BOOT WITH XA TRANSACTIONS	85
9.1. STATEFULSET RESOURCES	85
9.2. SPRING BOOT NARAYANA RECOVERY CONTROLLER	85
9.3. CONFIGURING SPRING BOOT NARAYANA RECOVERY CONTROLLER	85
9.4. RUNNING CAMEL SPRING BOOT XA QUICKSTART ON OPENSIFT	86
9.5. TESTING SUCCESSFUL XA TRANSACTIONS	88
9.6. TESTING FAILED XA TRANSACTIONS	88
CHAPTER 10. INTEGRATING A CAMEL APPLICATION WITH THE A-MQ BROKER	89
10.1. BUILDING AND DEPLOYING A SPRING BOOT CAMEL A-MQ QUICKSTART	89
CHAPTER 11. INTEGRATING SPRING BOOT WITH KUBERNETES	91
11.1. SPRING BOOT EXTERNALIZED CONFIGURATION	91
11.1.1. Kubernetes ConfigMap	91
11.1.2. Kubernetes Secrets	91
11.1.3. Spring Cloud Kubernetes plugin	91
11.1.4. Enabling Spring Boot with Kubernetes integration	91

11.2. RUNNING TUTORIAL FOR CONFIGMAP PROPERTY SOURCE	92
11.2.1. Running Spring Boot Camel Config quickstart	92
11.2.2. Configuration properties bean	94
11.2.3. Setting up Secret	96
11.2.4. Setting up ConfigMap	98
11.3. USING CONFIGMAP PROPERTYSOURCE	100
11.3.1. Applying individual properties	100
11.3.2. Applying application.yaml ConfigMap property	100
11.3.3. Applying application.properties ConfigMap property	100
11.3.4. Deploying a ConfigMap	101
11.4. USING SECRETS PROPERTYSOURCE	101
11.4.1. Example of setting Secrets	101
11.4.2. Consuming the Secrets	102
11.4.3. Configuration properties for Secrets PropertySource	103
11.5. USING PROPERTYSOURCE RELOAD	103
11.5.1. Enabling PropertySource Reload	103
11.5.2. Levels of PropertySource Reload	103
11.5.3. Example of PropertySource Reload	104
11.5.4. PropertySource Reload operating modes	105
11.5.5. PropertySource Reload configuration properties	105
CHAPTER 12. DEVELOPING AN APPLICATION FOR THE KARAF IMAGE	106
12.1. CREATING A KARAF PROJECT USING MAVEN ARCHETYPE	106
12.2. STRUCTURE OF THE CAMEL KARAF APPLICATION	107
12.3. KARAF ARCHETYPE CATALOG	108
12.4. USING FABRIC8 KARAF FEATURES	108
12.4.1. Adding Fabric8 Karaf features	108
12.4.2. Adding Fabric8 Karaf Core bundle functionality	109
12.4.3. Setting the Property Placeholder service options	109
12.4.4. Adding a custom property placeholder resolver	111
12.4.5. List of resolution strategies	112
12.4.6. List of Property Placeholder service options	112
12.5. ADDING FABRIC8 KARAF CONFIG ADMIN SUPPORT	113
12.5.1. Adding Fabric8 Karaf Config admin support	113
12.5.2. Adding ConfigMap injection	113
12.5.3. Configuration plugin	114
12.5.4. Config Property Placeholders	114
12.5.5. Fabric8 Karaf Config Admin options	114
12.6. ADDING FABRIC8 KARAF BLUEPRINT SUPPORT	115
12.7. ENABLING FABRIC8 KARAF HEALTH CHECKS	116
12.7.1. Configuring health checks	117
12.8. ADDING CUSTOM HEALTH CHECKS	118
CHAPTER 13. DEVELOPING AN APPLICATION FOR THE JBOSS EAP IMAGE	120
13.1. CREATING A JBOSS EAP PROJECT USING THE S2I SOURCE WORKFLOW	120
13.2. STRUCTURE OF THE JBOSS EAP APPLICATION	122
13.3. JBOSS EAP QUICKSTART TEMPLATES	122
CHAPTER 14. USING PERSISTENT STORAGE IN FUSE ON OPENSIFT	124
14.1. ABOUT VOLUMES AND VOLUME TYPES	124
14.2. ABOUT PERSISTENTVOLUMES	124
14.3. CONFIGURING PERSISTENT VOLUME	124
14.4. CREATING PERSISTENTVOLUMECLAIMS	125
14.5. USING PERSISTENT VOLUMES IN PODS	125

CHAPTER 15. PATCHING FUSE ON OPENSIFT	127
15.1. IMPORTANT NOTE ON BOMS AND MAVEN DEPENDENCIES	127
15.2. PATCHING THE FUSE ON OPENSIFT IMAGES	127
15.3. PATCHING THE FUSE ON OPENSIFT TEMPLATES	129
15.4. PATCH APPLICATION DEPENDENCIES USING BOM	129
15.4.1. Updating dependencies in a Spring Boot application	130
15.4.2. Updating dependencies in a Karaf application	131
15.4.3. Updating dependencies in a JBoss EAP application	132
15.5. AVAILABLE BOM VERSIONS	132
APPENDIX A. SPRING BOOT MAVEN PLUGIN	134
A.1. SPRING BOOT MAVEN PLUGIN GOALS	134
A.2. USING SPRING BOOT MAVEN PLUGIN	134
A.2.1. Using Spring Boot Maven plugin for Spring Boot 2	134
APPENDIX B. USING KARAF MAVEN PLUGIN	137
B.1. MAVEN DEPENDENCIES	137
B.2. KARAF MAVEN PLUGIN CONFIGURATION	137
B.3. CUSTOMIZED KARAF ASSEMBLY	138
B.3.1. karaf:assembly goal	138
APPENDIX C. OPENSIFT MAVEN PLUGIN	140
C.1. ABOUT OPENSIFT MAVEN PLUGIN	140
C.2. BUILDING IMAGES	140
C.3. KUBERNETES AND OPENSIFT RESOURCES	140
C.4. INSTALLING OPENSIFT MAVEN PLUGIN	141
C.5. UNDERSTANDING OPENSIFT MAVEN PLUGIN BUILD GOALS	142
C.6. UNDERSTANDING OPENSIFT MAVEN PLUGIN DEVELOPMENT GOALS	142
APPENDIX D. FABRIC8 MAVEN PLUGIN	144
D.1. BUILDING IMAGES	144
D.2. KUBERNETES AND OPENSIFT RESOURCES	144
D.3. INSTALLING THE PLUGIN	145
D.4. UNDERSTANDING FABRIC8 MAVEN PLUGIN GOALS	145
D.4.1. Understanding build and development goals	146
D.4.2. Setting environmental variable	146
D.4.3. Resource validation configuration	147
D.5. GENERATORS	147
D.5.1. Zero configuration	148
D.5.2. Modes for specifying the base image	148
D.5.2.1. Default values for istag mode	148
D.5.2.2. Default values for docker mode	149
D.5.2.3. Mode configuration for Spring Boot applications	149
D.5.2.4. Mode configuration for Karaf applications	149
D.5.2.5. Specifying the Generator mode using the command line	150
D.5.3. Spring Boot	150
D.5.4. Karaf	151
APPENDIX E. FABRIC8 CAMEL MAVEN PLUGIN	152
E.1. FABRIC8 CAMEL MAVEN PLUGIN GOALS	152
E.2. ADDING THE FABRIC8-CAMEL-MAVEN PLUGIN TO YOUR PROJECT	152
E.3. RUNNING THE GOAL ON ANY MAVEN PROJECT	153
E.4. OPTIONS	154
E.5. VALIDATING INCLUDE TEST	155

APPENDIX F. CUSTOMIZING JVM ENVIRONMENT VARIABLES	157
F.1. USING S2I JAVA BUILDER IMAGE WITH OPENJDK 8	157
F.2. USING S2I KARAF BUILDER IMAGE WITH OPENJDK 8	157
F.2.1. Configuring the Karaf4 assembly	157
F.2.2. Customizing the Maven build	157
F.3. BUILD TIME ENVIRONMENT VARIABLES	157
F.4. RUN TIME ENVIRONMENT VARIABLES	158
F.5. JOLOKIA CONFIGURATION	158
APPENDIX G. TUNING JVMs TO RUN IN LINUX CONTAINERS	160
G.1. TUNING THE JVM	160
G.2. DEFAULT BEHAVIOUR OF FUSE ON OPENSIFT IMAGES	160
G.3. CUSTOM TUNING OF FUSE ON OPENSIFT IMAGES	160
G.4. TUNING THIRD-PARTY LIBRARIES	161

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our [CTO Chris Wright's message](#).

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 with **registry.redhat.io**.
2. Installing the Fuse on OpenShift images and templates.

2.1. AUTHENTICATING WITH REGISTRY.REDHAT.IO FOR CONTAINER IMAGES

Configure authentication with **registry.redhat.io** before you can deploy Fuse container images on OpenShift.

Prerequisites

- Cluster administrator access to an OpenShift Container Platform cluster.
- OpenShift **oc** client tool is installed. For more details, see the [OpenShift CLI documentation](#).

Procedure

1. Log into your OpenShift cluster as administrator:

```
oc login --user system:admin --token=my-token --server=https://my-cluster.example.com:6443
```

2. Open the project in which you want to deploy Fuse:

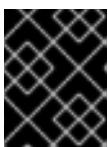
```
oc project myproject
```

3. Create a **docker-registry** secret using your Red Hat Customer Portal account, replacing **PULL_SECRET_NAME** with the secret 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
```

You should see the following output:

```
secret/PULL_SECRET_NAME created
```



IMPORTANT

You must create this **docker-registry** secret in every OpenShift project namespace that will authenticate to **registry.redhat.io**.

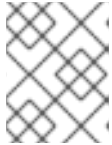
4. Link the secret to your service account to use the secret for pulling images. The following example uses the **default** service account:

```
oc secrets link default PULL_SECRET_NAME --for=pull
```

The service account name must match the name that the OpenShift pod uses.

5. Link the secret to the **builder** service account to use the secret for pushing and pulling build images:

```
oc secrets link builder PULL_SECRET_NAME
```



NOTE

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

Additional resources

For more details on authenticating with Red Hat for container images:

- [Red Hat container image authentication](#)
- [Red Hat registry service accounts](#)

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

OpenShift Container Platform 4.x 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 **registry.redhat.io**.
- 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:
  - fuse-console-rhel8
  - fuse-eap-openshift-jdk8-rhel7
  - fuse-eap-openshift-jdk11-rhel8
  - fuse-java-openshift-rhel8
  - fuse-java-openshift-jdk11-rhel8
  - fuse-karaf-openshift-rhel8
  - fuse-apicurito-generator-rhel8
  - fuse-apicurito-rhel8
  skippedTemplates:
  - s2i-fuse79-eap-camel-amq
  - s2i-fuse79-eap-camel-cdi
  - s2i-fuse79-eap-camel-cxf-jaxrs
  - s2i-fuse79-eap-camel-cxf-jaxws
  - s2i-fuse79-karaf-camel-amq
  - s2i-fuse79-karaf-camel-log
  - s2i-fuse79-karaf-camel-rest-sql
  - s2i-fuse79-karaf-cxf-rest
  - s2i-fuse79-spring-boot-2-camel-amq
  - s2i-fuse79-spring-boot-2-camel-config
  - s2i-fuse79-spring-boot-2-camel-drools
  - s2i-fuse79-spring-boot-2-camel-infinispan
  - s2i-fuse79-spring-boot-2-camel-rest-3scale
  - s2i-fuse79-spring-boot-2-camel-rest-sql
  - s2i-fuse79-spring-boot-2-camel
  - s2i-fuse79-spring-boot-2-camel-xa
  - s2i-fuse79-spring-boot-2-camel-xml
  - s2i-fuse79-spring-boot-2-cxf-jaxrs
```



```
- s2i-fuse79-spring-boot-2-cxf-jaxws
- s2i-fuse79-spring-boot-2-cxf-jaxrs-xml
- s2i-fuse79-spring-boot-2-cxf-jaxws-xml
```

7. Install Fuse on OpenShift image streams.

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005
```

```
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 \
karaf-camel-amq-template.json \
karaf-camel-log-template.json \
karaf-camel-rest-sql-template.json \
karaf-cxf-rest-template.json ;
do
oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/quickstarts/${template}
done
```

9. Install Spring Boot 2 quickstart templates:

```
for template in spring-boot-2-camel-amq-template.json \
spring-boot-2-camel-config-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-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 \
spring-boot-2-cxf-jaxrs-xml-template.json \
spring-boot-2-cxf-jaxws-xml-template.json ;
do oc create -n openshift -f \
```

```
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/quickstarts/${template}
done
```

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

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

2.3. INSTALLING API DESIGNER ON OPENSIFT 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 **registry.redhat.io**.

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 **Operators** and then click **OperatorHub**.
4. In the search field, type **API Designer**.
5. Click the **Red Hat Integration - API Designer** card. The **Red Hat Integration - API Designer** Operator install page opens.
6. Click **Install**. The **Install Operator** page opens.
 - a. For **Update Channel**, select **fuse-console-7.9.x**.
 - b. For **Installation mode**, select a namespace (project) from the list of namespaces on the cluster.
 - c. 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 **Install** 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 **Red Hat Integration - API Designer** in the list.

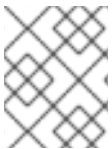
Next Steps

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 2.3.1, “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.

2.3.1. 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 **Red Hat Integration - API Designer**.
3. Under **Provided APIs**, click **Create instance**.
A default form with a minimal starting template for the API Designer instance opens. Accept the default values or, optionally, edit them.
4. Click **Create** to create a new **apicurito-service**. OpenShift starts up the pods, services, and other components for the new API Designer service.
5. 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.

6. To open the API Designer:
 - a. Select **Networking > Routes**.
 - b. Make sure that the correct project is selected.
 - c. In the **apicurito-service-ui** row's **Location** column, click the URL.
The API Designer web console opens in a new browser tab.

2.3.2. 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.



NOTE

The normal operator upgrade process does not work when upgrading from API Designer 7.8 to API Designer 7.9. To upgrade the API Designer from Fuse 7.8 to Fuse 7.9, you must delete the 7.8 API Designer operator and then install the 7.9 API Designer operator.

2.3.3. Metering labels for API Designer

You can use the OpenShift Metering operator to analyze your installed API Designer operator, UI component, and code generator to determine whether you are in compliance with your Red Hat subscription. For more information on Metering, see the [OpenShift documentation](#).

The following table lists the metering labels for the API Designer.

Table 2.1. API Designer Metering Labels

Label	Possible values
com.company	Red_Hat
rht.prod_name	Red_Hat_Integration
rht.prod_ver	7.9
rht.comp	Fuse
rht.comp_ver	7.9
rht.subcomp	fuse-apicurito apicurito-service-ui apicurito-service-generator

Label	Possible values
<code>rht.subcomp_t</code>	<code>infrastructure</code>

Examples

- Example for the API Designer **operator**:

```
apicurito-operator
com.company: Red_Hat
rht.prod_name: Red_Hat_Integration
rht.prod_ver: 7.9
rht.comp: Fuse
rht.comp_ver: 7.9
rht.subcomp: fuse-apicurito
rht.subcomp_t: infrastructure
```

- Example for the API Designer **UI** component:

```
com.company: Red_Hat
rht.prod_name: Red_Hat_Integration
rht.prod_ver: 7.9
rht.comp: Fuse
rht.comp_ver: 7.9
rht.subcomp: apicurito-service-ui
rht.subcomp_t: infrastructure
```

- Example for the API Designer **Generator** component:

```
com.company: Red_Hat
rht.prod_name: Red_Hat_Integration
rht.prod_ver: 7.9
rht.comp: Fuse
rht.comp_ver: 7.9
rht.subcomp: apicurito-service-generator
rht.subcomp_t: infrastructure
```

2.3.4. Considerations for installing API Designer in a restricted environment

The OpenShift clusters that are installed in a restricted environment, by default cannot access the Red Hat-provided OperatorHub sources because those remote sources require full Internet connectivity. In such environment, to install API designer operator, you must complete following prerequisites:

- Disable the default remote OperatorHub sources for Operator Lifecycle Manager (OLM).
- Use a workstation with full Internet access to create local mirrors of the OperatorHub content.
- Configure OLM to install and manage Operators from the local sources instead of the default remote sources.

For more information refer [Using Operator Lifecycle Manager on restricted networks](#) section in the OpenShift documentation. Once you have created local mirrors of the OperatorHub, you can perform next steps.

- Install API Designer using mirrored OperatorHub as per instructions described in the [Installing API Designer on OpenShift 4.x](#).
- Add API Designer as a service as per instructions described in the [Adding API Designer as a service to an OpenShift 4.x project](#)

2.4. SETTING UP THE FUSE CONSOLE ON OPENSIFT 4.X

On OpenShift 4.x, setting up the Fuse Console involves installing and deploying it. You have these options for installing and deploying the Fuse Console:

- [Section 2.4.1, "Installing and deploying the Fuse Console on OpenShift 4.x by using the OperatorHub"](#)
You can use the Fuse Console Operator to install and deploy the Fuse Console so that it has access to Fuse applications in a specific namespace. The Operator handles securing the Fuse Console for you.
- [Section 2.4.2, "Installing and deploying the Fuse Console on OpenShift 4.x by using the command line"](#)
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. You must secure the Fuse Console by generating a client certificate before you deploy it.

Optionally, you can customize role-based access control (RBAC) for the Fuse Console as described in [Section 2.4.3, "Role-based access control for the Fuse Console on OpenShift 4.x"](#) .

2.4.1. Installing and deploying the Fuse Console on OpenShift 4.x by using the OperatorHub

To install the Fuse Console on OpenShift 4.x, you can use the Fuse Console Operator provided in the OpenShift OperatorHub. To deploy the Fuse Console, you create an instance of the installed operator.

Prerequisites

- You have configured authentication with **registry.redhat.io** as described in [Authenticating with registry.redhat.io for container images](#).
- If you want to customize role-based access control (RBAC) for the Fuse Console, you must have a RBAC configuration map file in the same OpenShift namespace to which you install the Fuse Console Operator. If you want to use the default RBAC behavior, as described in [Role-based access control for the Fuse Console on OpenShift 4.x](#), you do not need to provide a configuration map file.

Procedure


To install and deploy the Fuse Console:

1. Log in to the OpenShift console in your web browser as a user with **cluster admin** access.
2. Click **Operators** and then click **OperatorHub**.
3. In the search field window, type **Fuse Console** to filter the list of operators.
4. Click **Fuse Console Operator**.

5. In the Fuse Console Operator install window, click **Install**.
The **Create Operator Subscription** form opens.
 - For **Update Channel**, select **7.9.x**.
 - For **Installation Mode**, accept the default (a specific namespace on the cluster).
Note that after you install the operator, when you deploy the Fuse Console, you can choose to monitor applications in all namespaces on the cluster or to monitor applications only in the namespace in which the Fuse Console operator is installed.
 - For **Installed Namespace**, select the namespace in which you want to install the Fuse Console Operator.
 - For the **Approval Strategy**, you can select **Automatic** or **Manual** to configure how OpenShift handles updates to the Fuse Console Operator.
 - If you select **Automatic** updates, when a new version of the Fuse Console Operator is available, the OpenShift Operator Lifecycle Manager (OLM) automatically upgrades the running instance of the Fuse Console 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 Fuse Console Operator updated to the new version.
6. Click **Install**.
OpenShift installs the Fuse Console Operator in the current namespace.
7. To verify the installation, click **Operators** and then click **Installed Operators**. You can see the Fuse Console in the list of operators.
8. To deploy the Fuse Console by using the OpenShift web console:
 - a. In the list of **Installed Operators**, under the **Name** column, click **Fuse Console**.
 - b. On the **Operator Details** page under **Provided APIs**, click **Create Instance**.
Accept the configuration default values or optionally edit them.

For **Replicas**, if you want to increase the Fuse Console performance (for example, in a high availability environment), you can increase the number of pods allocated to the Fuse Console.

For **Rbac** (role-based access control), only specify a value in the **config Map** field if you want to customize the default RBAC behavior and if the ConfigMap file already exists in the namespace in which you installed the Fuse Console Operator. For more information about RBAC, see [Role-based access control for the Fuse Console on OpenShift 4.x](#) .
 - c. Click **Create**.
The **Fuse Console Operator Details** page opens and shows the status of the deployment.
9. To open the Fuse Console:
 - a. For a **namespace** deployment: In the OpenShift web console, open the project in which you installed the Fuse Console operator, and then select **Overview**. In the **Project Overview** page, scroll down to the **Launcher** section and click the Fuse Console URL to open it.

For a **cluster** deployment, in the OpenShift web console's title bar, click the grid icon (). In the popup menu, under **Red Hat applications**, click the Fuse Console URL link.

- b. Log into the Fuse Console.
An **Authorize Access** page opens in the browser listing the required permissions.
 - c. Click **Allow selected permissions**.
The Fuse Console opens in the browser and shows the Fuse application pods that you have authorization to access.
10. Click **Connect** for the application that you want to view.
A new browser window opens showing the application in the Fuse Console.

2.4.2. 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.
- **cluster with role-based access control**- The cluster template with configurable role-based access control (RBAC). For more information, see [Role-based access control for the Fuse Console on OpenShift 4.x](#).
- **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.
- **namespace with role-based access control**- The namespace template with configurable RBAC. For more information, see [Role-based access control for the Fuse Console on OpenShift 4.x](#).

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-2.1.0.fuse-sb2-790047-redhat-00005/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.
- You have configured authentication with **registry.redhat.io** as described in [Authenticating with registry.redhat.io for container images](#).
- 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
```

- 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/fuse-console-rhel8:1.9 --from=registry.redhat.io/fuse7/fuse-console-rhel8:1.9 --confirm -n openshift
```

- Obtain the Fuse Console **APP_NAME** value by running the following command:

```
oc process --parameters -f TEMPLATE-FILENAME
```

where **TEMPLATE-FILENAME** is one of the following templates:

- Cluster template:
<https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-cluster-os4.json>
- Cluster template with configurable RBAC:
<https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-cluster-rbac.yml>
- Namespace template:
<https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-namespace-os4.json>
- Namespace template with configurable RBAC:
<https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-namespace-rbac.yml>

For example, for the cluster template with configurable RBAC, run this command:

```
oc process --parameters -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-cluster-rbac.yml
```

- 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
```

- 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 {templates-base-url}/fuse-console-cluster-os4.json -p ROUTE_HOSTNAME=myhost"
```

- For the cluster with RBAC template:

```
oc new-app -n myproject -f {templates-base-url}/fuse-console-cluster-rbac.yml -p
ROUTE_HOSTNAME=myhost"
```

- For the namespace template:

```
{templates-base-url}/fuse-console-namespace-os4.json
```

- For the namespace with RBAC template:

```
oc new-app -n myproject -f {templates-base-url}/fuse-console-namespace-rbac.yml
```

6. To configure the Fuse Console so that it can open the OpenShift Web console, set the **OPENSHIFT_WEB_CONSOLE_URL** environment variable by running the following command:

```
oc set env dc/${APP_NAME} OPENSHIFT_WEB_CONSOLE_URL=`oc get -n openshift-
config-managed cm console-public -o jsonpath={.data.consoleURL}`
```

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

```
oc status
```

8. To access the Fuse Console from a browser, use the URL that is returned in Step 7 (for example, <https://fuse-console.192.168.64.12.nip.io>).

2.4.2.1. 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, a client certificate must be generated before the Fuse Console is deployed. The service signing certificate authority private key must be used to sign the client certificate.

You must follow this procedure **only** if you are installing and deploying the Fuse Console by using the command line. If you are using the Fuse Console Operator, it handles this task for you.



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 **easypki**, **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
```

Here, the values in the **CN** parameter refers to the application name and the namespace that the application uses.

- 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](#).

2.4.3. Role-based access control for the Fuse Console on OpenShift 4.x

The Fuse Console offers role-based access control (RBAC) that infers access according to the user authorization provided by OpenShift. In the Fuse Console, RBAC determines a user's ability to perform MBean operations on a pod.

For information on OpenShift authorization see the [Using RBAC to define and apply permissions](#) section of the OpenShift documentation.

Role-based access is enabled by default when you use the Operator to install the Fuse Console on OpenShift.

If you want to implement role-based access for the Fuse Console by installing it with a template, you must use one of the templates that are configurable with RBAC (**`fuse-console-cluster-rbac.yml`** or **`fuse-console-namespace-rbac.yml`**) to install the Fuse Console as described in [Installing and deploying the Fuse Console on OpenShift 4.x by using the command line](#).

Fuse Console RBAC leverages the user's **verb** access on a pod resource in OpenShift to determine the user's access to a pod's MBean operations in the Fuse Console. By default, there are two user roles for the Fuse Console:

- **admin**
If a user can **update** a pod in OpenShift, then the user is conferred the **admin** role for the Fuse Console. The user can perform **write** MBean operations in the Fuse Console for the pod.
- **viewer**
If a user can **get** a pod in OpenShift, then the user is conferred the **viewer** role for the Fuse Console. The user can perform **read-only** MBean operations in the Fuse Console for the pod.



NOTE

If you used a non-RBAC template to install the Fuse Console, only OpenShift users that are granted the **update** verb on the pod resource are authorized to perform the Fuse Console MBeans operations. Users that are granted the **get** verb on the pod resource can **view** the pod but they cannot perform any Fuse Console operations.

Additional resources

- [Determining access roles for the Fuse Console on OpenShift 4.x](#)
- [Customizing role-based access to the Fuse Console on OpenShift 4.x](#)
- [Disabling role-based access control for the Fuse Console on OpenShift 4.x](#)

2.4.3.1. Determining access roles for the Fuse Console on OpenShift 4.x

The Fuse Console role-based access control is inferred from a user's OpenShift permissions for a pod. To determine the Fuse Console access role granted to a particular user, obtain the OpenShift permissions granted to the user for a pod.

Prerequisites

- You know the user's name.
- You know the pod's name.

Procedure

- To determine whether a user has the Fuse Console **admin** role for the pod, run the following command to see whether the user can update the pod on OpenShift:

```
oc auth can-i update pods/<pod> --as <user>
```

If the response is **yes**, the user has the Fuse Console **admin** role for the pod. The user can perform **write** MBean operations in the Fuse Console for the pod.

- To determine whether a user has the Fuse Console **viewer** role for the pod, run the following command to see whether the user can **get** a pod on OpenShift:

```
oc auth can-i get pods/<pod> --as <user>
```

If the response is **yes**, the user has the Fuse Console **viewer** role for the pod. The user can perform **read-only** MBean operations in the Fuse Console for the pod. Depending on the context, the Fuse Console prevents the user with the **viewer** role from performing a **write** MBean operation, by disabling an option or by displaying an "operation not allowed for this user" message when the user attempts a **write** MBean operation.

If the response is **no**, the user is not bound to any Fuse Console roles and the user cannot view the pod in the Fuse Console.

Additional resources

- [Role-based access control for the Fuse Console on OpenShift 4.x](#)
- [Customizing role-based access to the Fuse Console on OpenShift 4.x](#)
- [Disabling role-based access control for the Fuse Console on OpenShift 4.x](#)

2.4.3.2. Customizing role-based access to the Fuse Console on OpenShift 4.x

If you use the OperatorHub to install the Fuse Console, role-based access control (RBAC) is enabled by default as described in [Role-based access control for the Fuse Console on OpenShift 4.x](#). If you want to customize the Fuse Console RBAC behavior, before you deploy the Fuse Console, you must provide a ConfigMap file (that defines the custom RBAC behavior). You must place the custom ConfigMap file in the same namespace in which you installed the Fuse Console Operator.

If you use the command line templates to install the Fuse Console, the **deployment-cluster-rbac.yml** and **deployment-namespace-rbac.yml** templates create a ConfigMap that contains the configuration file (**ACL.yml**). The configuration file defines the roles allowed for MBean operations.

Prerequisite

- You installed the Fuse Console by using the OperatorHub or by using one of the Fuse Console RBAC templates (**deployment-cluster-rbac.yml** or **deployment-namespace-rbac.yml**)

Procedure

To customize the Fuse Console RBAC roles:

1. If you installed the Fuse Console by using the command line, the installation templates include a default ConfigMap file and so you can skip to the next step.

If you installed the Fuse Console by using the OperatorHub, before you deploy the Fuse Console create a RBAC ConfigMap:

- a. Make sure the current OpenShift project is the project to which you want to install the Fuse Console. For example, if you want to install the Fuse Console in the **fusetest** project, run this command:

```
oc project fusetest
```

- b. To create a Fuse Console RBAC ConfigMap file from a template, run this command:

```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/2.1.x.sb2.redhat-7-8-x/fuse-console-operator-rbac.yml -p APP_NAME=fuse-console | oc create -f -
```

2. Open the ConfigMap in an editor by running the following command:

```
oc edit cm $APP_NAME-rbac
```

For example:

```
oc edit cm fuse-console-rbac
```

3. Edit the file.
4. Save the file to apply the changes. OpenShift automatically restarts the Fuse Console pod.

Additional resources

- [Role-based access control for the Fuse Console on OpenShift 4.x](#)
- [Determining access roles for the Fuse Console on OpenShift 4.x](#)
- [Disabling role-based access control for the Fuse Console on OpenShift 4.x](#)

2.4.3.3. Disabling role-based access control for the Fuse Console on OpenShift 4.x

If you installed the Fuse Console by using the command line and you specified one of the Fuse Console RBAC templates, the Fuse Console's **HAWTIO_ONLINE_RBAC_ACL** environment variable passes the role-based access control (RBAC) ConfigMap configuration file path to the OpenShift server. If the **HAWTIO_ONLINE_RBAC_ACL** environment variable is not specified, RBAC support is disabled and only users that are granted the **update** verb on the pod resource (in OpenShift) are authorized to call MBeans operations on the pod in the Fuse Console.

Note that when you use the OperatorHub to install the Fuse Console, role-based access is enabled by default and the **HAWTIO_ONLINE_RBAC_ACL** environment variable does not apply.

Prerequisite

You installed the Fuse Console by using the command line and you specified one of the Fuse Console RBAC templates (**deployment-cluster-rbac.yml** or **deployment-namespace-rbac.yml**).

Procedure

To disable role-based access for the Fuse Console:

1. In OpenShift, edit the **Deployment Config** resource for the Fuse Console.
2. Delete the entire **HAWTIO_ONLINE_RBAC_ACL** environment variable definition. (Note that only clearing its value is not sufficient).
3. Save the file to apply the changes. OpenShift automatically restarts the Fuse Console pod.

Additional resources

- [Role-based access control for the Fuse Console on OpenShift 4.x](#)
- [Determining access roles for the Fuse Console on OpenShift 4.x](#)
- [Customizing role-based access to the Fuse Console on OpenShift 4.x](#)

2.4.4. 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, depending on how the application is configured.

For Fuse Console applications, you can also trigger an upgrade to an application 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:

1. 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"}}'
```

2. 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
```

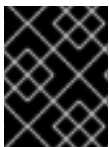
A successful deployment shows this response:

```
deployment "example-fuseconsole" successfully rolled out
```

2.5. CONFIGURING PROMETHEUS TO MONITOR FUSE APPLICATIONS ON OPENSIFT

2.5.1. About Prometheus

[Prometheus](#) is an open-source systems and service monitoring and alerting toolkit that you can use to monitor services deployed in your Red Hat OpenShift environment. Prometheus collects and stores metrics from configured services at given intervals, evaluates rule expressions, displays the results, and can trigger alerts if a specified condition becomes true.



IMPORTANT

Red Hat support for Prometheus is limited to the setup and configuration recommendations provided in Red Hat product documentation.

To monitor OpenShift services, you must configure each service to expose an endpoint to Prometheus format. This endpoint is an HTTP interface that provides a list of metrics and the current values of the metrics. Prometheus periodically scrapes each target-defined endpoint and writes the collected data to its database. Prometheus gathers data over an extended time, rather than just for the currently running session. Prometheus stores the data so that you can graphically visualize and run queries on the data.

2.5.1.1. Prometheus queries

In the Prometheus web interface, you can write queries in [Prometheus Query Language \(PromQL\)](#) to select and aggregate the collected data.

For example, you can use the following query to select all of the values that Prometheus has recorded within the last five minutes for all time-series data that has **http_requests_total** as the metric name:

```
http_requests_total[5m]
```

To further define or filter the results of the query, specify a label (a **key:value** pair) for the metric. For example, you can use the following query to select all the values that Prometheus has recorded within the last five minutes for all time-series data that has the metric name **http_requests_total** and a job label set to **integration**:


```
http_requests_total{job="integration"}[5m]
```

2.5.1.2. Options for displaying Prometheus data

You can specify how Prometheus handles the result of a query:

- View Prometheus data as tabular data in Prometheus’s expression browser.
- Consume Prometheus data by external systems through the [Prometheus HTTP API](#).
- Display Prometheus data in a graph.
Prometheus provides a default graphical view of the data that it collects. If you prefer a more robust graphical dashboard to view Prometheus data, Grafana is a popular choice.



NOTE

Grafana is a community-supported feature. Deploying Grafana to monitor Red Hat products is not supported with Red Hat production service level agreements (SLAs).

You can also use the PromQL language to configure alerts in [Prometheus’s Alertmanager tool](#).

2.5.2. Setting up Prometheus

To set up Prometheus, install the Prometheus operator custom resource definition on the cluster and then add Prometheus to an OpenShift project that includes a Fuse application.

Prerequisites

- You have **cluster admin** access to the OpenShift cluster.
- You have prepared the OpenShift cluster by installing the Fuse on OpenShift images and templates as described in the [Fuse on OpenShift Guide](#).
- You have created an OpenShift project on the cluster and added a Fuse application to it.

Procedure

1. Login to OpenShift with administrator permissions:

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

2. Install the custom resource definitions necessary for running the Prometheus operator:

```
oc create -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-prometheus-crd.yml
```

The Prometheus operator is now available to any namespace on the cluster.

3. Install the Prometheus operator to your namespace by using the following command syntax:

```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-prometheus-operator.yml -p NAMESPACE=<YOUR NAMESPACE> | oc create -f -
```

For example, use this command for a project (namespace) named **myproject**:

```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-prometheus-operator.yml -p NAMESPACE=myproject | oc create -f -
```



NOTE

The first time that you install the Prometheus operator into a namespace, it might take a few minutes for the Prometheus resource pods to start. Subsequently, if you install it to other namespaces on your cluster, the Prometheus resource pods start much faster.

4. Instruct the Prometheus operator to monitor the Fuse application in the project by using the following command syntax::

```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-servicemonitor.yml -p NAMESPACE=<YOUR NAMESPACE> -p FUSE_SERVICE_NAME=<YOUR FUSE SERVICE> | oc apply -f -
```

For example, use this command for an OpenShift project (namespace) named **myproject** that includes a Fuse application named **myfuseapp**:

```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-servicemonitor.yml -p NAMESPACE=myproject -p FUSE_SERVICE_NAME=myfuseapp | oc apply -f -
```

5. To open the Prometheus dashboard:
 - a. Login to the OpenShift console.
 - b. Open the project to which you added Prometheus.
 - c. In the left pane, select **Applications** -> **Routes**.

OPENSIFT CONTAINER PLATFORM

My Project

Routes [Learn More](#)

Name	Hostname	Service	Target Port	TLS Termination
addressbook-api	http://addressbook-api-myproject.192.168.64.49.nip.io	addressbook-api	web	
prometheus	http://prometheus-myproject.192.168.64.49.nip.io	prometheus		

- d. Click the Prometheus Hostname URL to open the Prometheus dashboard in a new browser tab or window.

Prometheus Alerts Graph Status Help

Enable query history

- insert metric at cursor -

Graph Console

Element	Value
no data	

- e. For information about getting started with Prometheus, go to: https://prometheus.io/docs/prometheus/latest/getting_started/

2.5.3. OpenShift environment variables

To configure your application's Prometheus instance, you can set the OpenShift environment variables listed in [Table 2.2, "Prometheus Environment Variables"](#).

Table 2.2. Prometheus Environment Variables

Environment Variable	Description	Default
AB_PROMETHEUS_HOST	The host address to bind.	0.0.0.0
AB_PROMETHEUS_OFF	If set, disables the activation of Prometheus (echoes an empty value).	Prometheus is enabled.
AB_PROMETHEUS_PORT	The Port to use.	9779

Environment Variable	Description	Default
AB_JMX_EXPORTER_CONFIG	Uses the file (including path) as the Prometheus configuration file.	The <code>/opt/prometheus/prometheus-config.yml</code> file with Camel metrics.
AB_JMX_EXPORTER_OPTS	Additional options to append to the JMX exporter configuration.	Not applicable.

Additional resources

For information on setting environment variables for a pod, see the *OpenShift Developer Guide* (https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html/developer_guide/).

2.5.4. Controlling the metrics that Prometheus monitors and collects

By default, Prometheus uses a configuration file (<https://raw.githubusercontent.com/jboss-fuse/application-templates/master/prometheus/prometheus-config.yml>) that includes all possible metrics exposed by Camel.

If you have custom metrics within your application that you want Prometheus to monitor and collect (for example, the number of orders that your application processes), you can use your own configuration file. Note that the metrics that you can identify are limited to those supplied in JMX.

Procedure

To use a custom configuration file to expose JMX beans that are not covered by the default Prometheus configuration, follow these steps:

1. Create a custom Prometheus configuration file. You can use the contents of the default file (**prometheus-config.yml** <https://raw.githubusercontent.com/jboss-fuse/application-templates/master/prometheus/prometheus-config.yml>) as a guide for the format. You can use any name for the custom configuration file, for example: **my-prometheus-config.yml**.
2. Add your prometheus configuration file (for example, **my-prometheus-config.yml**) to your application's **src/main/jkube-includes** directory.
3. Create a **src/main/jkube/deployment.xml** file within your application and add an entry for the **AB_JMX_EXPORTER_CONFIG** environment variable with its value set to your configuration file. For example:

```
spec:
  template:
    spec:
      containers:
      -
        resources:
          requests:
            cpu: "0.2"
          limits:
            cpu: "1.0"
```

```
env:
- name: SPRING_APPLICATION_JSON
  value: '{"server":{"tomcat":{"max-threads":1}}}'
- name: AB_JMX_EXPORTER_CONFIG
  value: "my-prometheus-config.yml"
```

This environment variable applies to your application at the pod level.

4. Rebuild and deploy your application.

2.6. USING METERING FOR FUSE ON OPENSIFT

You can use the Metering tool that is available on OCP 4 to generate metering reports from different data sources. As a cluster administrator, you can use metering to analyze what is happening in your cluster. You can either write your own, or use predefined SQL queries to define how you want to process data from the different data sources you have available. Using Prometheus as a default data source, you can generate reports on pods, namespaces, and most other Kubernetes resources. You must install and configure the Metering operator on OpenShift Container Platform 4.x first to use the Metering tool. For more information on Metering, see [Metering](#).

2.6.1. Metering resources

Metering has many resources which can be used to manage the deployment and installation of metering, as well as the reporting functionality metering provides. Metering is managed using the following CustomResourceDefinitions (CRDs):

Table 2.3. Metering resources

Name	Description
MeteringConfig	Configures the metering stack for deployment. Contains customizations and configuration options to control each component that makes up the metering stack.
Reports	Controls what query to use, when, and how often the query should be run, and where to store the results.
ReportQueries	Contains the SQL queries used to perform analysis on the data contained within ReportDataSources.
ReportDataSources	Controls the data available to ReportQueries and Reports. Allows configuring access to different databases for use within metering.

2.6.2. Metering labels for Fuse on OpenShift

Table 2.4. Metering Labels

Label	Possible values
com.company	Red_Hat

Label	Possible values
rht.prod_name	Red_Hat_Integration
rht.prod_ver	7.9
rht.comp	Fuse
rht.comp_ver	7.9
rht.subcomp	fuse7-java-openshift fuse7-eap-openshift fuse7-karaf-openshift
rht.subcomp_t	infrastructure

2.7. MONITORING FUSE ON OPENSIFT WITH CUSTOM GRAFANA DASHBOARDS

OpenShift Container Platform 4.6 provides monitoring dashboards that help you understand the state of cluster components and user-defined workloads.

Prerequisites

- You must have installed and deployed Prometheus on your cluster. Refer <https://github.com/jboss-fuse/application-templates/blob/master/monitoring/prometheus.md> for more information on how to install Grafana on OpenShift 4.
- You must have installed and configured Grafana.

Custom Dashboards for Fuse on OpenShift

There are two custom dashboards that you can use for Fuse on OpenShift. To use these dashboards, you must have installed and configured Grafana and Prometheus on your cluster. There are two kinds of example dashboards provided for Fuse on OpenShift. You can import these dashboards from [Fuse Grafana dashboards](#).

- **Fuse Pod / Instance Metrics Dashboard:**
This dashboard collects metrics from a single Fuse application pod / instance. You can import the dashboard using **fuse-grafana-dashboard.yml**. The table of panels for the Fuse Pod metrics dashboard on OpenShift includes:

Table 2.5. Fuse Pod metrics dashboard

Title	Legend	Query	Description
-------	--------	-------	-------------

Title	Legend	Query	Description
Process Start Time	-	process_start_time_seconds{pod="\$pod"}*1000	Time when the process started
Current Memory HEAP	-	sum(jvm_memory_bytes_used{pod="\$pod", area="heap"})*100/sum(jvm_memory_bytes_max{pod="\$pod", area="heap"})	Memory currently being used by Fuse
Memory Usage	committed	sum(jvm_memory_bytes_committed{pod="\$pod"})	Memory committed
	used	sum(jvm_memory_bytes_used{pod="\$pod"})	Memory used
	max	sum(jvm_memory_bytes_max{pod="\$pod"})	Maximum memory
Threads	current	jvm_threads_current{pod="\$pod"}	Number of current threads
	daemon	jvm_threads_daemon{pod="\$pod"}	Number of daemon threads
	peak	jvm_threads_peak{pod="\$pod"}	Number of peak threads
Camel Exchanges / 1m	Exchanges Completed / 1m	sum(increase(org_apache_camel_Exchanges_Completed{pod="\$pod"}[1m]))	Completed Camel exchanges per minute
	Exchanges Failed / 1m	sum(increase(org_apache_camel_Exchanges_Failed{pod="\$pod"}[1m]))	Failed Camel exchanges per minute
	Exchanges Total / 1m	sum(increase(org_apache_camel_Exchanges_Total{pod="\$pod"}[1m]))	Total Camel exchanges per minute

Title	Legend	Query	Description
	Exchanges Inflight	<code>sum(org_apache_camel_ExchangesInflight{pod="\$pod"})</code>	Camel exchanges currently being processed
Camel Processing Time	Delta Processing Time	<code>sum(org_apache_camel_DeltaProcessingTime{pod="\$pod"})</code>	Delta of Camel processing time
	Last Processing Time	<code>sum(org_apache_camel_LastProcessingTime{pod="\$pod"})</code>	Last Camel processing time
	Max Processing Time	<code>sum(org_apache_camel_MaxProcessingTime{pod="\$pod"})</code>	Maximum Camel processing time
	Min Processing Time	<code>sum(org_apache_camel_MinProcessingTime{pod="\$pod"})</code>	Minimum Camel processing time
	Mean Processing Time	<code>sum(org_apache_camel_MeanProcessingTime{pod="\$pod"})</code>	Mean Camel processing time
Camel Service Durations	Maximum Duration	<code>sum(org_apache_camel_MaxDuration{pod="\$pod"})</code>	Maximum Camel service durations
	Minimum Duration	<code>sum(org_apache_camel_MinDuration{pod="\$pod"})</code>	Minimum Camel service durations
	Mean Duration	<code>sum(org_apache_camel_MeanDuration{pod="\$pod"})</code>	Mean Camel service durations
Camel Failures & Redeliveries	Redeliveries	<code>sum(org_apache_camel_Redeliveries{pod="\$pod"})</code>	Number of redeliveries
	Last Processing Time	<code>sum(org_apache_camel_LastProcessingTime{pod="\$pod"})</code>	Last Camel processing time
	External Redeliveries	<code>sum(org_apache_camel_ExternalRedeliveries{pod="\$pod"})</code>	Number of external redeliveries

- Fuse Camel Route Metrics Dashboard:

This dashboard collects metrics from a single Camel route in a Fuse application. You can import the dashboard using **fuse-grafana-dashboard-routes.yml**. The table of panels for the Fuse Camel Route metrics dashboard on OpenShift includes:

Table 2.6. Fuse Camel Route metrics dashboard

Title	Legend	Query	Description
Exchanges per second	-	<code>rate(org_apache_camel_ExchangesTotal{route="\\$route\""}[5m])</code>	Total Camel exchanges per second
Exchanges inflight	-	<code>max(org_apache_camel_ExchangesInflight{route="\\$route\""})</code>	Number of Camel exchanges currently being processed
Exchanges failure rate	-	<code>sum(org_apache_camel_ExchangesFailed{route="\\$route\""}) / sum(org_apache_camel_ExchangesTotal{route="\\$route\""})</code>	Percentage of failed Camel exchanges
Mean processing time	-	<code>org_apache_camel_MeanProcessingTime{route="\\$route\""}[5m]</code>	Mean Camel processing time
Exchanges per second	Failed	<code>rate(org_apache_camel_ExchangesFailed{route="\\$route\""}[5m])</code>	Failed exchanges per second
	Completed	<code>rate(org_apache_camel_ExchangesCompleted{route="\\$route\""}[5m])</code>	Completed exchanges per second
Exchanges inflight	Exchanges inflight	<code>org_apache_camel_ExchangesInflight{route="\\$route\""}[5m]</code>	Camel exchanges currently being processed
Processing time	Max	<code>org_apache_camel_MaxProcessingTime{route="\\$route\""}[5m]</code>	Maximum Camel processing time
	Mean	<code>org_apache_camel_MeanProcessingTime{route="\\$route\""}[5m]</code>	Mean Camel processing time

Title	Legend	Query	Description
	Min	org_apache_camel_MinProcessingTime{route="\\$route\""}}	Minimum Camel processing time
External Redeliveries per second	-	rate(org_apache_camel_ExternalRedeliveries{route="\\$route\""}[5m])	External redeliveries per second
Redeliveries per second	-	rate(org_apache_camel_Redeliveries{route="\\$route\""}[5m])	Redeliveries per second
Failures handled per second	-	rate(org_apache_camel_FailuresHandled{route="\\$route\""}[5m])	Failures handled per second

2.8. INSTALLING FUSE IMAGESTREAMS AND TEMPLATES ON THE OPENSIFT 3.X SERVER

After you configure authentication to **registry.redhat.io**, 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-2.1.0.fuse-sb2-790047-redhat-00005
```

```
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 \
```

```

karaf-camel-amq-template.json \
karaf-camel-log-template.json \
karaf-camel-rest-sql-template.json \
karaf-cxf-rest-template.json ;
do
oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-
2.1.0.fuse-sb2-790047-redhat-00005/quickstarts/${template}
done

```

6. Install Spring Boot 2 quickstart templates:

```

for template in spring-boot-2-camel-amq-template.json \
spring-boot-2-camel-config-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-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 \
spring-boot-2-cxf-jaxrs-xml-template.json \
spring-boot-2-cxf-jaxws-xml-template.json ;
do oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-
2.1.0.fuse-sb2-790047-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-2.1.0.fuse-sb2-790047-redhat-00005/fis-console-cluster-
template.json
oc create -n openshift -f https://raw.githubusercontent.com/jboss-fuse/application-
templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/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

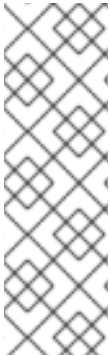
```

2.8.1. Setting up the Fuse Console on OpenShift 3.11

On OpenShift 3.11, you can access the Fuse Console:

- By adding the Fuse Console to an OpenShift project so that you can monitor all the running Fuse containers in the project.
- By adding the Fuse Console to an OpenShift cluster so that you can monitor all the running Fuse containers in all projects on the cluster.
- By opening it from a specific Fuse pod so that you can monitor that single running Fuse container.

You deploy the Fuse Console templates from the command line.



NOTE

To install Fuse Console on Minishift or CDK based environments, follow the steps explained in the KCS article below.

- To install Fuse Console on Minishift or CDK based environments, see [KCS 4998441](#).
- If it is necessary to disable Jolokia authentication see the workaround described in [KCS 3988671](#).

Prerequisite

- Install the Fuse on OpenShift image streams and the templates for the Fuse Console as described in [Fuse on OpenShift Guide](#).



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 3.11.

[Section 2.8.1.1, "Deploying the Fuse Console on OpenShift 3.11"](#)

[Section 2.8.1.2, "Monitoring a single Fuse pod from the Fuse Console on OpenShift 3.11"](#)

2.8.1.1. Deploying the Fuse Console on OpenShift 3.11

[Table 2.7, "Fuse Console templates"](#) describes the OpenShift 3.11 templates that you can use to deploy the Fuse Console from the command line, depending on the type of Fuse application deployment.

Table 2.7. 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.

Type	Description
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-2.1.0.fuse-sb2-790047-redhat-00005/fis-console-namespace-template.json
```



NOTE

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.

Prerequisite

- 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
```

Procedure

To deploy the Fuse Console from the command line:

- 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-2.1.0.fuse-sb2-790047-redhat-00005/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-2.1.0.fuse-sb2-790047-redhat-00005/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>).

2.8.1.2. 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.

Prerequisite

- 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"
          }
        ]
      }
    ]
  }
}
```

Procedure

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:

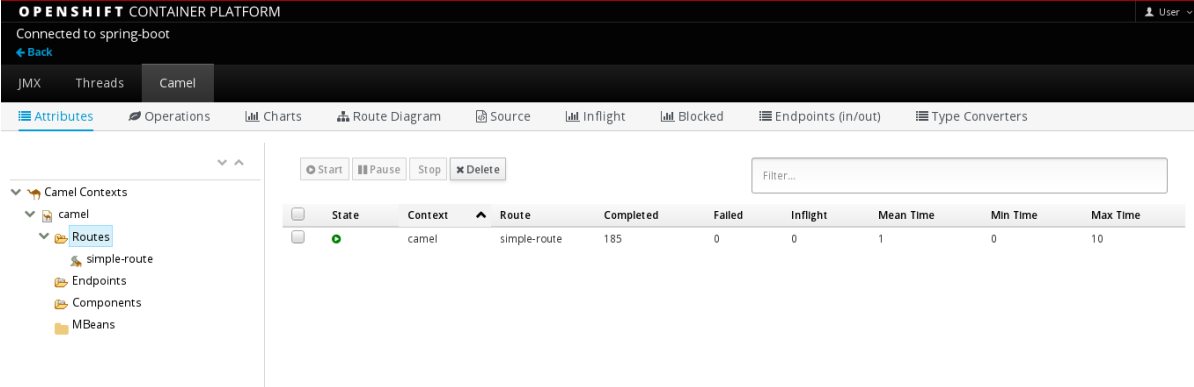
Template

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 left sidebar shows a tree view of Camel contexts, with "Routes" expanded to show "simple-route". The main content area displays a table of Camel routes with the following data:

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

CHAPTER 3. INSTALLING FUSE ON OPENSIFT IN A RESTRICTED ENVIRONMENT

To install Fuse on OpenShift in a non-restricted environment, you pull imagestreams and templates from **registry.redhat.io**. In a production environment which has no or limited internet access, that is not possible. This section explains how to install Fuse on OpenShift in a restricted environment.

Prerequisites

- You have installed and configured OpenShift server so that it can run in a restricted environment.

3.1. SETTING UP INTERNAL DOCKER REGISTRY

This section explains how to set up internal docker registry which can be used to push or pull images. You must configure an internal docker registry where you can pull or push images.

Procedure

1. Install internal ROOT CA.

```
cd /etc/pki/ca-trust/source/anchors
sudo curl -O https://password.corp.redhat.com/RH-IT-Root-CA.crt
sudo update-ca-trust extract
sudo update-ca-trust update
```

This certificate allows the system to authenticate itself to the registry.

2. Login to **registry.redhat.io**.

```
docker login -u USERNAME -p PASSWORD registry.redhat.io
```

3. Pull the Fuse on OpenShift images from **registry.redhat.io**.

```
docker pull registry.redhat.io/fuse7/fuse-java-openshift-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-java-openshift-jdk11-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-karaf-openshift-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-console-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-apicurito-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-apicurito-generator-rhel8:1.9
```

4. Tag the pulled imagestreams.

```
docker tag registry.redhat.io/fuse7/fuse-java-openshift-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse7/fuse-java-openshift-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-java-openshift-jdk11-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse7/fuse-java-openshift-jdk11-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-karaf-openshift-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse-karaf-openshift-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-console-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse7-fuse-console-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-apicurito-rhel8:1.9 docker-
```



```
registry.upshift.redhat.com/fuse7-fuse-apicurito-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-apicurito-generator-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse7-fuse-apicurito-generator-rhel8:1.9
```

5. Push the tagged imagestreams to the internal docker registry.

```
docker push docker-registry.upshift.redhat.com/fuse7/fuse-java-openshift-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse7/fuse-java-openshift-jdk11-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse-karaf-openshift-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse7-fuse-console-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse7-fuse-apicurito-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse7-fuse-apicurito-generator-rhel8:1.9
```

3.2. CONFIGURING INTERNAL REGISTRY SECRETS

After setting up the restricted docker registry and pushing all the images, it is necessary to configure the restricted OpenShift server so that it can communicate with the internal registry.

Procedure

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

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

2. 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 psi-internal-registry <pull_secret_name> \
--docker-server=docker-registry.upshift.redhat.com \
--docker-username=CUSTOMER_PORTAL_USERNAME \
--docker-password=CUSTOMER_PORTAL_PASSWORD \
--docker-email=EMAIL_ADDRESS
```

3. 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.

```
oc secrets add serviceaccount/builder secrets/psi-internal-registry
oc secrets add serviceaccount/default secrets/psi-internal-registry --for=pull
oc secrets add serviceaccount/builder secrets/psi-internal-registry
```

4. 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 default psi-internal-registry
oc secrets link default psi-internal-registry --for=pull
oc secrets link builder psi-internal-registry
```

3.3. INSTALLING FUSE ON OPENSIFT IMAGES IN A RESTRICTED ENVIRONMENT

The **fis-image-streams.json** file contains the imageStream definitions for Red Hat Fuse on OpenShift. But, all the imagestreams refer to **registry.redhat.io**. You must change all the **registry.redhat.io** references to the **psi-internal-registry** URL.

Procedure

1. Download Red Hat Fuse on OpenShift imagestream json file.

```
curl -o fis-image-streams.json {BASEURL}
```

2. Open the **fis-image-streams.json** file and locate all the references to **registry.redhat.io**. For example:

```
{
  "name": "1.9",
  "annotations": {
    "description": "Red Hat Fuse 7.9 Karaf S2I images.",
    "openshift.io/display-name": "Red Hat Fuse 7.9 Karaf",
    "iconClass": "icon-rh-integration",
    "tags": "builder,jboss-fuse,java,karaf,xpaas,hidden",
    "supports": "jboss-fuse:7.9.0,java:8,xpaas:1.2",
    "version": "1.9"
  },
  "referencePolicy": {
    "type": "Local"
  },
  "from": {
    "kind": "DockerImage",
    "name": "registry.redhat.io/fuse7/fuse-karaf-openshift-rhel8:1.9"
  }
},
```

3. Replace all the **registry.redhat.io** references in the file with **psi-internal-registry** name. For example:

```
{
  "name": "1.9",
  "annotations": {
    "description": "Red Hat Fuse 7.9 Karaf S2I images.",
    "openshift.io/display-name": "Red Hat Fuse 7.9 Karaf",
    "iconClass": "icon-rh-integration",
    "tags": "builder,jboss-fuse,java,karaf,xpaas,hidden",
    "supports": "jboss-fuse:7.9.0,java:8,xpaas:1.2",
    "version": "1.9"
  },
  "referencePolicy": {
    "type": "Local"
  },
  "from": {
    "kind": "DockerImage",
    "name": "docker-registry.upshift.redhat.com/fuse7/fuse-karaf-openshift-rhel8:1.9"
  }
},
```

4. After all the references are replaced, run the following command to install Fuse on OpenShift imagestreams:

```
oc create -f fis-image-streams.json -n {namespace}
```

3.4. USING AN INTERNAL MAVEN REPOSITORY

In a restricted environment, you need to use a different Maven Repository. You can specify it using a template parameter named **MAVEN_MIRROR_URL**. You can use this **MAVEN_MIRROR_URL** parameter to create a new application from command line.

3.4.1. Running a Spring Boot application with MAVEN_MIRROR_URL

This example explains how to deploy and run a Spring Boot Application using MAVEN_MIRROR_URL.

Procedure

1. Download the Spring Boot Camel XML quickstart.

```
oc create -f {BASEURL}/quickstarts/spring-boot-2-camel-xml-template.json
```

2. Enter the following command to create the resources required for running the Spring Boot quickstart template using the **MAVEN_MIRROR_URL** parameter. 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-fuse79-spring-boot-2-camel-xml -n {namespace} -p
IMAGE_STREAM_NAMESPACE={namespace} -p MAVEN_MIRROR_URL={Maven mirror
URL}
```

3.4.2. Running a Spring Boot application with OpenShift Maven plugin

This example explains how to deploy and run a Spring Boot application with OpenShift Maven plugin using internal Maven repository.

Procedure

1. To run the quickstart with OpenShift Maven plugin, download the Spring Boot 2 camel archetype from local repository and then deploy the quickstart. Replace **{Maven Mirror URL}** with the Maven mirror repository URL.

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
-DarchetypeCatalog={Maven Mirror URL}/archetypes/archetypes-catalog/2.2.0.fuse-sb2-
790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-redhat-00004-archetype-
catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-xml-archetype
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004
```

2. 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': : fuse79-spring-boot2
```

```
Define value for property 'version': 1.0-SNAPSHOT: :  
Define value for property 'package': org.example.fis: :  
Confirm properties configuration:  
groupId: org.example.fis  
artifactId: fuse79-spring-boot  
version: 1.0-SNAPSHOT  
package: org.example.fis  
Y: : Y
```

3. If the above command exited with the BUILD SUCCESS status, you should now have a new Fuse on OpenShift project under the **fuse79-spring-boot2** subdirectory.
4. You are now ready to build and deploy the **fuse79-spring-boot2** project. Assuming you are still logged into OpenShift, change to the directory of the **fuse79-spring-boot2** project, and then build and deploy the project, as follows.

```
cd fuse79-spring-boot2  
mvn oc:deploy -Popenshift
```

CHAPTER 4. 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.

4.1. 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 with **registry.redhat.io**.

For more information see:

- [Authenticating with **registry.redhat.io** for container images](#)
- [Red Hat CDK 3.16 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-2.1.0.fuse-sb2-790047-redhat-00005
```

```
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.

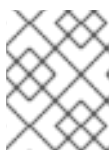
```
for template in eap-camel-amq-template.json \
eap-camel-cdi-template.json \
eap-camel-cxf-jaxrs-template.json \
eap-camel-cxf-jaxws-template.json \
karaf-camel-amq-template.json \
karaf-camel-log-template.json \
karaf-camel-rest-sql-template.json \
karaf-cxf-rest-template.json ;
do
oc create -n test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

5. Install Spring Boot 2 quickstart templates:

```
for template in spring-boot-2-camel-amq-template.json \
spring-boot-2-camel-config-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-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 \
spring-boot-2-cxf-jaxrs-xml-template.json \
spring-boot-2-cxf-jaxws-xml-template.json ;
do oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/quickstarts/${template}
done
```

6. 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](#).

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

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

8. In your browser, navigate to the OpenShift console:
 - a. Use https://OPENSIFT_IP_ADDR:8443 and replace **OPENSIFT_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 5. GETTING STARTED FOR DEVELOPERS

5.1. 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](#)

5.1.1. 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 7.9. 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 3.16 of CDK. Detailed instructions for installing and using CDK 3.16 are provided in the [Red Hat CDK 3.16 Getting Started Guide](#).
2. Configure your OpenShift credentials to gain access to the Red Hat Ecosystem Catalog by following the instructions in [Authenticating with registry.redhat.io for container images](#).
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 3.16 Getting Started Guide](#).

5.1.2. 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.

5.1.3. 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):

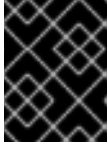
```
$ minishift oc-env
export PATH="/Users/john/.minishift/cache/oc/v1.5.0:$PATH"
# Run this command to configure your shell:
# eval $(minishift oc-env)
```

For more details, see [Using the OpenShift Client Binary](#) in CDK 3.16 *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 3.16 *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](#).

5.1.4. 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.

5.2. 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 *Builds OpenShift Container Platform 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 *Builds OpenShift Container Platform guide*.

5.2.1. 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.



NOTE

Running quickstarts with JDK11

Use the correct JDK11 profile during the compile time if you want to use JDK11 based image at runtime. When building and deploying the quickstarts using JDK11, ensure that you have installed JDK11 on your build machine and then build your quickstarts using the correct JDK11 profile.

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-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-xml-archetype \
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004
```

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

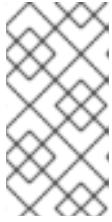
```
Define value for property 'groupId': : org.example.fis
Define value for property 'artifactId': : fuse79-spring-boot
Define value for property 'version': : 1.0-SNAPSHOT :
Define value for property 'package': : org.example.fis :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse79-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse79-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 **fuse79-spring-boot** subdirectory. You can inspect the XML DSL code in the **fuse79-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.

- 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 images and templates are already installed and you can access them.

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

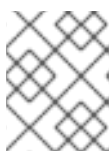
If the images and templates are not pre-installed, or if the provided versions are out of date, install (or update) the Fuse on OpenShift images and templates manually. For more information on how to install Fuse on OpenShift images see [Chapter 2, Getting Started for administrators](#).

- You are now ready to build and deploy the **fuse79-spring-boot** project. Assuming you are still logged into OpenShift, change to the directory of the **fuse79-spring-boot** project, and then build and deploy the project, as follows.

```
cd fuse79-spring-boot
mvn oc:deploy -Popenshift
```

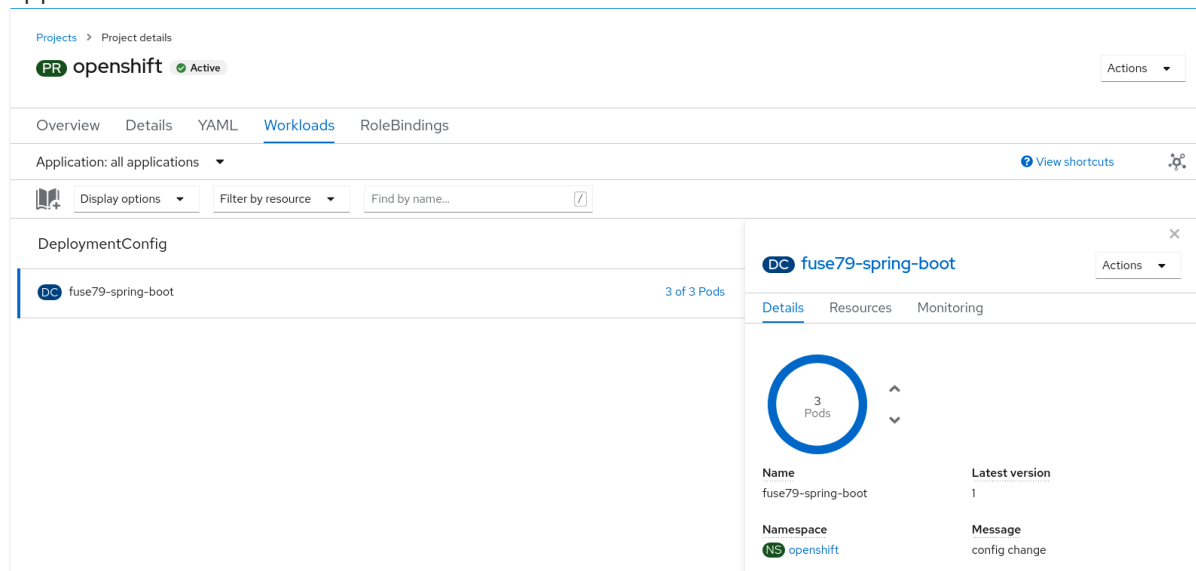
At the end of a successful build, you should see some output like the following.

```
...
[INFO] OpenShift platform detected
[INFO] Using project: openshift
[INFO] Creating a Service from openshift.yml namespace openshift name fuse79-spring-boot
[INFO] Created Service: target/jkube/applyJson/openshift/service-fuse79-spring-boot.json
[INFO] Using project: openshift
[INFO] Creating a DeploymentConfig from openshift.yml namespace openshift name fuse79-
spring-boot
[INFO] Created DeploymentConfig: target/jkube/applyJson/openshift/deploymentconfig-
fuse79-spring-boot.json
[INFO] Creating Route openshift:fuse79-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: 2020-12-04T12:15:06+05:30
[INFO] Final Memory: 63M/688M
[INFO] -----
```

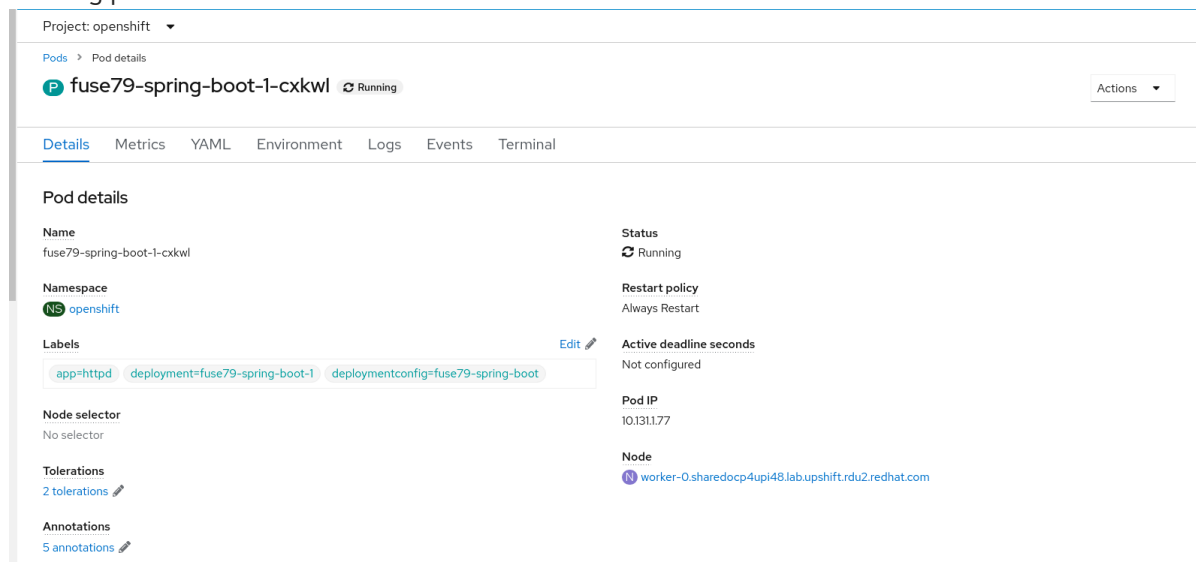
**NOTE**

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 **fuse79-spring-boot** to view the Overview information page for the **fuse79-spring-boot** application.




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, **fuse79-spring-boot-xxxxx**) to view the details of the running pod.



13. Click on the **Logs** tab to view the application log and scroll down the log to find the 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 **fuse79-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.

5.2.2. Undeploying and redeploying the project

You can undeploy or redeploy your projects, as follows:

Procedure

- To undeploy the project, enter the command:

```
mvn oc:undeploy
```

- To redeploy the project, enter the commands:

```
mvn oc:undeploy
mvn oc:deploy -Popenshift
```

5.2.3. 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.

- 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 7.9 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-fuse7-spring-boot-camel-xml

--> Deploying template "openshift/s2i-fuse7-spring-boot-camel-xml" to project openshift
...
--> Creating resources ...
  imagestream.image.openshift.io "s2i-fuse7-spring-boot-camel-xml" created
  buildconfig.build.openshift.io "s2i-fuse7-spring-boot-camel-xml" created
  deploymentconfig.apps.openshift.io "s2i-fuse7-spring-boot-camel-xml" created
--> Success
  Build scheduled, use 'oc logs -f bc/s2i-fuse7-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-fuse7-spring-boot-camel-xml** to view the **Overview** information page for the quickstart.

Projects > Project Details

PR openshift Active Actions

Overview Details YAML **Workloads** Role Bindings

Display Options Filter by Resource Find by name...

Deployment Config

DC s2i-fuse78-spring-boot-2-camel-xml 1 of 1 pods

DC s2i-fuse78-spring-boot-2-camel-xml Actions

Details Resources Monitoring

1 pod

Name Latest Version
s2i-fuse78-spring-boot-2-camel-xml 1

Namespace Message
NS openshift config change

Labels Update Strategy
a...=s2i-fuse78-spring-boot-2-c... Rolling

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

Builds > Build Details

B s2i-fuse78-spring-boot-2-camel-xml-1 Complete

Details YAML Environment **Logs** Events

Some lines have been abridged because they are exceptionally long.
To view unabridged log content, you can either [open the raw file in another window](#) or [download it](#).

Log stream ended.

```

1001 lines
Pushing image image-registry.openshift-image-registry.svc:5000/openshift/s2i-fuse78-spring-boot-2-camel-xml:latest ...
Pushing image "image-registry.openshift-image-registry.svc:5000/openshift/s2i-fuse78-spring-boot-2-camel-xml:latest" from local storage.
Setting authentication secret for "".
Getting image source signatures
Copying blob sha256:d1441e19f34168e1c1d9958b64ebb5a04daf336251373908b574245bf2498309
Copying blob sha256:c10d665fbb802a300b8c94c3120b144cc5be1d7b36222b390c7c35b374bce8a4
Copying blob sha256:f691f2730256bcf1bbcddaf53f261a6fb263c746229eb8741e75a59e6a15200e
Copying blob sha256:522d9d05816a63661e850202de4da07e0871c23dec7132069458651f57e96188
Copying blob sha256:9605c76aba31e1dc1b56d2a08a06eab3019a6465d804a91f300747058cf72d57
Copying config sha256:a89545de5a4d9e02a9094532e448ab3e197758b24adac0adc723e71e4618646d

```

- In the left hand side panel, expand **Workloads**.
- Click **Pods** and then click **s2i-fuse7-spring-boot-camel-xml-xxxx**. The pod details for the application are displayed.

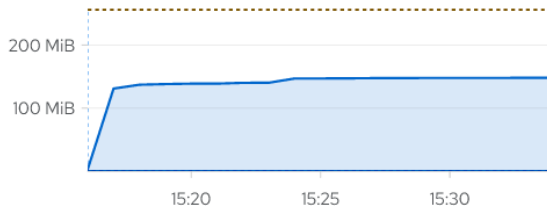
Pods > Pod Details

P s2i-fuse78-spring-boot-2-camel-xml-1-czrx2 Running

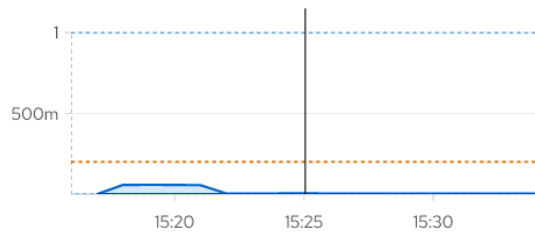
[Details](#) [YAML](#) [Environment](#) [Logs](#) [Events](#) [Terminal](#)

Pod Details

Memory Usage



CPU Usage




Network In



Network Out



11. To shut down the running pod,
 - a. On the Project Status page for the **openshift** project, click **s2i-fuse7-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 6. DEVELOPING AN APPLICATION FOR THE SPRING BOOT IMAGE

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

6.1. 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-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-xml-archetype \
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004
```

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': : fuse79-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: fuse79-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse79-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 **fuse79-spring-boot** subdirectory.
4. You are now ready to build and deploy the **fuse79-spring-boot** project. Assuming you are still logged into OpenShift, change to the directory of the **fuse79-spring-boot** project, and then build and deploy the project, as follows.

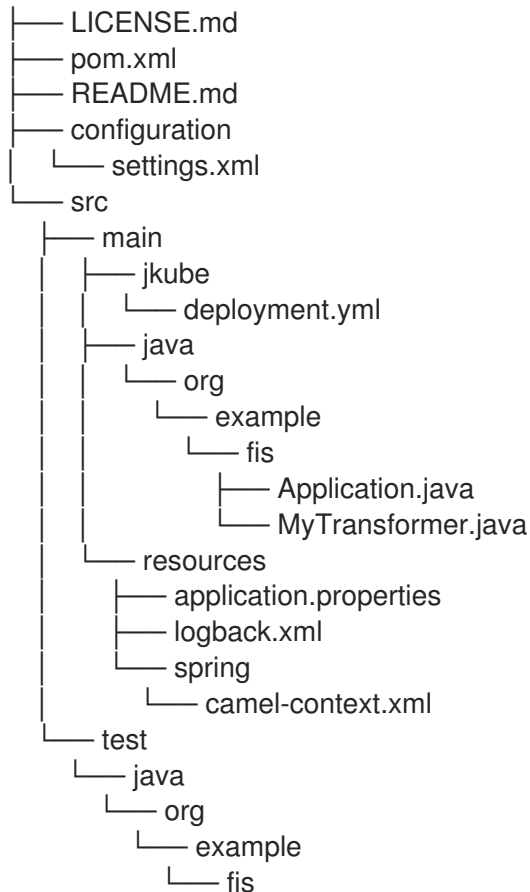
```
cd fuse79-spring-boot
mvn oc:deploy -Popenshift
```

**NOTE**

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

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

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

Application.java

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

camel-context.xml

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



NOTE

You can find more information on developing Spring-Boot applications at [Developing your first Spring Boot Application](#)

src/main/jkube/deployment.yml

Provides additional configuration that is merged with the default OpenShift configuration file generated by the openshift-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.

6.3. SPRING BOOT 2 ARCHETYPE CATALOG

The Spring Boot 2 Archetype catalog includes the following examples.

Table 6.1. 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.
spring-boot-camel-xml-archetype	Demonstrates how to configure Camel routes in Spring Boot via a Spring XML configuration file.
spring-boot-cxf-jaxrs-archetype	Demonstrates how to use Apache CXF with Spring Boot based on a fabric8 Java base image. The quickstart uses Spring Boot to configure an application that includes a CXF JAXRS endpoint with Swagger enabled.
spring-boot-cxf-jaxws-archetype	Demonstrates how to use Apache CXF with Spring Boot based on a fabric8 Java base image. The quickstart uses Spring Boot to configure an application that includes a CXF JAXWS endpoint.

Name	Description
spring-boot-cxf-jaxrs-xml-archetype	Demonstrates how to use Apache CXF JAX-RS with Spring Boot 2 on OpenShift. This quickstart uses Spring Boot2 to launch a Spring configuration file based CXF application that includes a CXF JAXRS endpoint with Swagger enabled.
spring-boot-cxf-jaxws-xml-archetype	Demonstrates how to use Apache CXF JAX-WS with Spring Boot 2 on OpenShift. The quickstart uses Spring Boot2 to launch a Spring configuration file based CXF 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>
```

6.4. BOM FILE FOR SPRING BOOT

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

IMPORTANT

Ensure you are using the correct Fuse BOM based on the version of Spring Boot you are using.

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.

6.5. 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:

- [Spring Boot 2 BOM](#)

Spring Boot 2 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>7.9.0.fuse-sb2-790065-redhat-00001</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 is 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.

6.6. 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.

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>7.9.0.fuse-sb2-790065-redhat-00001</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 7. 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.

7.1. 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.

7.2. 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 7.3, "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](#).

7.3. 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-openapi-java** (**camel-openapi-java-starter** is included)

7.4. 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
```

7.5. 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);  
    }  
  
}
```

7.6. 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");  
            }  
  
        };  
    }  
  
}
```

7.7. 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}");
    }
}
```

7.8. 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.

7.9. 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
```

7.10. 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
```

7.11. 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);
    }
}

```

7.12. 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**.

7.13. 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
```

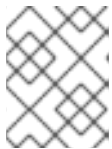
7.14. ADDING XML ROUTES TO THE CLASSPATH FOR AUTO-CONFIGURATION

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
```

7.15. 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>
```

7.16. 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 8. RUNNING SOAP TO REST BRIDGE QUICKSTART FOR SPRING BOOT 2 ON FUSE ON OPENSIFT

This quickstart demonstrates how to use Camel's REST DSL to expose a backend SOAP API. A simple camel route can bridge REST invocation to legacy SOAP service. Security is involved for both REST endpoint and SOAP endpoint, both backed by RH SSO. Frontend REST API protected via OAuth and OpenID Connect, and the client will fetch JWT access token from RH SSO using **Resource Owner Password Credentials** OAuth2 mode and using this token to access the REST endpoint.

Prerequisites

- You have installed and configured OCP 3.11 or later version.
- You have installed RH SSO 7.4 or later version.
- You have installed 3Scale 2.8 or later version.
- You have configured authentication to **registry.redhat.io**. For more information see [Configuring Red Hat Container Registry authentication](#).

Procedure

Following section explains how to run and deploy SOAP to REST bridge quickstart on Fuse on OpenShift.

1. Start OpenShift server. Since we need to install RH SSO image (2 pods) and 3Scale image (15 pods) as prerequisites for this quickstart, we need to start the OpenShift server on a powerful machine, with options **--memory 8GB --cpus 4** We also need to issue a security token with the expiration time, hence we need to add the timezone option as well. Ensure the Openshift cluster uses the same time zone as your local machine (by default it will use UTC timezone).
2. Add **cluster-admin** role to the user **developer**.

```
$ oc login -u system:admin
$ oc adm policy add-cluster-role-to-user cluster-admin developer
$ oc login -u developer
$ oc project openshift
```

This quickstart is deployed in the **openshift** namespace (this is the requirement of default configurations of the templates involved), as well as the RH SSO image, so we need to add the **cluster-admin** role to user **developer**.

3. Create a secret and link it to the **serviceaccounts**.

```
$ oc create secret docker-registry camel-bridge --docker-server=registry.redhat.io \
--docker-username=USERNAME \
--docker-password=PASSWORD \
--docker-email=EMAIL_ADDRESS
$ oc secrets link default camel-bridge --for=pull
$ oc secrets link builder camel-bridge
```

4. Add the RH SSO image stream and install RH SSO with template **sso74-x509-postgresql-persistent**.

```
$ for resource in sso74-image-stream.json \
```

```

sso74-https.json \
sso74-postgresql.json \
sso74-postgresql-persistent.json \
sso74-x509-https.json \
sso74-x509-postgresql-persistent.json
do
  oc create -f \
  https://raw.githubusercontent.com/jboss-container-images/redhat-sso-7-openshift-
image/sso74-dev/templates/${resource}
done

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

$ oc new-app --template=sso74-x509-postgresql-persistent

```

Verify that the RH SSO images are available from **openshift** namespace, and then install RH SSO with template **sso74-x509-postgresql-persistent**. This template can save the RH SSO configuration permanently, so the configuration is retained after the OpenShift server restart.

5. Once the RH SSO image is installed successfully on the server, you can see the output on the console as follows.

A new persistent RH-SSO service (using PostgreSQL) has been created in your project. The admin username/password for accessing the master realm via the RH-SSO console is tprYtXP1/nEjf7fojv11FmhJ5eaqadoh0SI2gvlls. The username/password for accessing the PostgreSQL database "root" is userqxe/XNYRjL74CrJEWW7HiSYEdH5FMKVSDytx. The HTTPS keystore used for serving secure content, the JGroups keystore used for securing JGroups communications, and server truststore used for securing RH-SSO requests were automatically created via OpenShift's service serving x509 certificate secrets.

- * With parameters:
 - * Application Name=sso
 - * Custom RH-SSO Server Hostname=
 - * JGroups Cluster Password=1whGRnsAWu162u0e4P6jNpLn5ysJLWjg # generated
 - * Database JNDI Name=java:jboss/datasources/KeycloakDS
 - * Database Name=root
 - * Datasource Minimum Pool Size=
 - * Datasource Maximum Pool Size=
 - * Datasource Transaction Isolation=
 - * PostgreSQL Maximum number of connections=
 - * PostgreSQL Shared Buffers=
 - * Database Username=userqxe # generated
 - * Database Password=XNYRjL74CrJEWW7HiSYEdH5FMKVSDytx # generated
 - * Database Volume Capacity=1Gi
 - * ImageStream Namespace=openshift
 - * RH-SSO Administrator Username=tprYtXP1 # generated
 - * RH-SSO Administrator Password=nEjf7fojv11FmhJ5eaqadoh0SI2gvlls # generated
 - * RH-SSO Realm=
 - * RH-SSO Service Username=
 - * RH-SSO Service Password=
 - * PostgreSQL Image Stream Tag=10
 - * Container Memory Limit=1Gi

6. Note down the Username/Password which is used to access the RH SSO admin console. For example,

- * RH-SSO Administrator Username=tpYtXP1 # generated
- * RH-SSO Administrator Password=nEjf7fojv11FmhJ5eaqadoh0Sl2gvlls # generated

7. Install 3scale template in the 3scale project.

```
$ oc new-project 3scale
$ oc create secret docker-registry threescale-registry-auth --docker-server=registry.redhat.io
--docker-server=registry.redhat.io \
--docker-username=USERNAME \
--docker-password=PASSWORD \
--docker-email=EMAIL_ADDRESS
$ oc secrets link default threescale-registry-auth --for=pull
$ oc secrets link builder threescale-registry-auth
$ oc new-app --param WILDCARD_DOMAIN="OPENSIFT_IP_ADDR.nip.io" -f
https://raw.githubusercontent.com/3scale/3scale-amp-openshift-
templates/2.8.0.GA/amp/amp-eval-tech-preview.yml
```

3scale installation on openshift will start 15 pods, so it is necessary to create a new specific project for 3scale. You also need a new **threescale-registry-auth** (use this name to create the secret as it is written in 3scale templates) secret for 3scale. You can reuse the USERNAME/PASSWORD from camel-bridge secret. We intentionally use **amp-eval-tech-preview.yml** template here because it doesn't explicitly specify hardware resources so can be easily run on a local machine/laptop.

8. After the 3scale template is installed successfully on the Openshift, you can see the output on the console as follows.

3scale API Management

```
-----
3scale API Management main system (Evaluation)

Login on https://3scale-admin.192.168.64.33.nip.io as admin/b6t784nt

* With parameters:
* AMP_RELEASE=2.8
* APP_LABEL=3scale-api-management
* TENANT_NAME=3scale
* RWX_STORAGE_CLASS=null
* AMP_BACKEND_IMAGE=registry.redhat.io/3scale-amp2/backend-rhel7:3scale2.8
* AMP_ZYNC_IMAGE=registry.redhat.io/3scale-amp2/zync-rhel7:3scale2.8
* AMP_APICAST_IMAGE=registry.redhat.io/3scale-amp2/apicast-gateway-
rhel8:3scale2.8
* AMP_SYSTEM_IMAGE=registry.redhat.io/3scale-amp2/system-rhel7:3scale2.8
* ZYNC_DATABASE_IMAGE=registry.redhat.io/rhsc/postgresql-10-rhel7
* MEMCACHED_IMAGE=registry.redhat.io/3scale-amp2/memcached-rhel7:3scale2.8
* IMAGESTREAM_TAG_IMPORT_INSECURE=false
* SYSTEM_DATABASE_IMAGE=registry.redhat.io/rhsc/mysql-57-rhel7:5.7
* REDIS_IMAGE=registry.redhat.io/rhsc/redis-32-rhel7:3.2
* System MySQL User=mysql
* System MySQL Password=mrschf4h # generated
* System MySQL Database Name=system
* System MySQL Root password.=xbi0ch3i # generated
* WILDCARD_DOMAIN=192.168.64.33.nip.io
* SYSTEM_BACKEND_USERNAME=3scale_api_user
* SYSTEM_BACKEND_PASSWORD=kraji167 # generated
```

```

* SYSTEM_BACKEND_SHARED_SECRET=8af5m6gb # generated
*
SYSTEM_APP_SECRET_KEY_BASE=726e63427173e58cbb68a63bdc60c7315565d6acd037c
aedeeb0050ecc0e6e41c3c7ec4aba01c17d8d8b7b7e3a28d6166d351a6238608bb84aa5d5b2d
c02ae60 # generated
* ADMIN_PASSWORD=b6t784nt # generated
* ADMIN_USERNAME=admin
* ADMIN_EMAIL=
* ADMIN_ACCESS_TOKEN=k055jof4itblvwwn # generated
* MASTER_NAME=master
* MASTER_USER=master
* MASTER_PASSWORD=buikudum # generated
* MASTER_ACCESS_TOKEN=xa7wkt16 # generated
* RECAPTCHA_PUBLIC_KEY=
* RECAPTCHA_PRIVATE_KEY=
* SYSTEM_REDIS_URL=redis://system-redis:6379/1
* SYSTEM_MESSAGE_BUS_REDIS_URL=
* SYSTEM_REDIS_NAMESPACE=
* SYSTEM_MESSAGE_BUS_REDIS_NAMESPACE=
* Zync Database PostgreSQL Connection Password=efyJdRccBbYcWtWI # generated
* ZYNC_SECRET_KEY_BASE=dcmNGWtrjCReuJIQ # generated
* ZYNC_AUTHENTICATION_TOKEN=3FKMAije3V3RWQQ8 # generated
* APICAST_ACCESS_TOKEN=2ql8txu4 # generated
* APICAST_MANAGEMENT_API=status
* APICAST_OPENSSL_VERIFY=false
* APICAST_RESPONSE_CODES=true
* APICAST_REGISTRY_URL=http://apicast-staging:8090/policies

```

9. Note down the Username/Password which can access the 3scale admin console.

```

* ADMIN_PASSWORD=b6t784nt # generated
* ADMIN_USERNAME=admin

```

10. Configure RH SSO.

- a. Login to RH SSO Admin Console from https://sso-openshift.OPENSHIFT_IP_ADDR.nip.io/auth with username/password displayed on console after the RH SSO installation.
- b. Click the **Add Realm** button on the upper left corner of the page.
- c. On the **Add Realm** page, select Import **Select file** button.
- d. Select `./src/main/resources/keycloak-config/realm-export-new.json` from the directory which will import pre-defined necessary **realm/client/user/role** for this example.

11. Configure 3Scale API Gateway.

- a. Login to 3Scale Admin Console from https://3scale-admin.OPENSHIFT_IP_ADDR.nip.io/p/admin/dashboard with username/password displayed on console after the 3Scale installation.
- b. When creating a new product, select **Define manually** and use **camel-security-bridge** for both **Name** and **System name**.

- c. When creating a new backend, use **camel-security-bridge** for both **Name** and **System name** and the **Private Base URL** should be http://spring-boot-camel-soap-rest-bridge-openshift.OPENSIFT_IP_ADDR.nip.io/.
- d. Add the newly created backend to the newly created product.
- e. Add the Mapping Rule **Verb:POST Pattern:/**.
- f. When creating application plans, use **camel-security-bridge** for both **Name** and **System name**.
- g. When creating applications, choose the new created **camel-security-bridge** application plan. After creating the application, note down the API Credentials. Use these credentials to access the 3scale gateway. For example,

```
User Key bdfb53fe9b426fbf21428fd116035798
```

- h. Edit the newly created **camel-security-bridge** project and publish it from **camel-security-bridge** in the Dashboard.
 - i. Go to Integration > Settings. Select **As HTTP Headers** as the **Credentials location**.
 - j. From the **camel-security-bridge** in the Dashboard, go to Integration > Configuration and promote both the **Staging APIcast** and **Production APIcast**.
12. Navigate to the directory that contains the extracted quickstart application (for example, my_openshift/spring-boot-camel-soap-rest-bridge).

```
$ cd my_openshift/spring-boot-camel-soap-rest-bridge
```

13. Build and deploy the project to the OpenShift cluster.

```
$ mvn clean oc:deploy -Popenshift -DJAVA_OPTIONS="-Dso.server=https://sso-openshift.OPENSIFT_IP_ADDR.nip.io -Dweather.service.host=${your local ip}"
```

We need to pass in two properties to **camel-soap-rest-bridge** image on openshift. One is the RH SSO server address on openshift, and this is https://sso-openshift.OPENSIFT_IP_ADDR.nip.io. Another one is the backend soap server. In this quickstart, we run the backend soap server on the local machine, so pass the local ip address of your machine as `-Dweather.service.host`. (This must be an ip address other than localhost or 127.0.0.1).

14. In your browser, navigate to the **openshift** project in the OpenShift console. Wait until you can see that the pod for the **spring-boot-camel-soap-rest-bridge** has started up.
15. On the project's Overview page, navigate to the details page deployment of the **spring-boot-camel-soap-rest-bridge** application:
https://OPENSIFT_IP_ADDR:8443/console/project/openshift/browse/pods/spring-boot-camel-soap-rest-bridge-NUMBER_OF_DEPLOYMENT?tab=details.
16. Switch to **Logs** tab to view the log from Camel.
17. Access OpenApi API.

This example provides API documentation of the service using openapi using the context-path camelxf/openapi. You can access the API documentation from your Web browser at <http://spring->

[boot-camel-soap-rest-bridge-](#)
[openshift.OPENSIFT_IP_ADDR.nip.io/camelcxf/openapi/openapi.jsonn.](#)

CHAPTER 9. 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.

9.1. 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.

9.2. 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.

9.3. 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.

9.4. 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-7.9.0.fuse-sb2-790055-redhat-00002
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=POSTGRES_USER=theuser --
param=POSTGRES_PASSWORD='Thepassword1!' --
env=POSTGRES_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 oc:deploy -Popenshift
```

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})
```

9.5. 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.

9.6. 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 10. INTEGRATING A CAMEL APPLICATION WITH THE A-MQ BROKER

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

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

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

Prerequisites

- Ensure that AMQ Broker is installed and running as described in [Deploying AMQ Broker 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. Create a new project for quickstart, for example:.

```
oc new-project quickstart
```

3. Retrieve the quickstart project by using the Maven archetype:

```
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-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml -DarchetypeGroupId=org.jboss.fuse.fis.archetypes -
DarchetypeArtifactId=spring-boot-camel-amq-archetype -DarchetypeVersion=2.2.0.fuse-sb2-
790047-redhat-00004
```

4. Navigate to the quickstart directory **fuse79-spring-boot-camel-amq**.

```
cd fuse79-spring-boot-camel-amq
```

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

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

```
oc apply -f src/main/resources/k8s
```

6. Create the ConfigMap for the application, for example:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: spring-boot-camel-amq-config
  namespace: quickstarts
data:
  service.host: 'fuse-broker-amqps-0-svc'
  service.port.amqp: '5672'
  service.port.amqps: '5671'
```

7. Run the **mvn** command to deploy the quickstart to the OpenShift server, by using the ImageStream from Step 3:

```
mvn oc:deploy -Popenshift -Dkubernetes.generator.fromMode=istag -
Dkubernetes.generator.from=openshift/fuse-java-openshift:1.9
```

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 **fuse79-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 **fuse79-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 11. 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](#)

11.1. 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).

11.1.1. 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.

11.1.2. Kubernetes Secrets

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

11.1.3. 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.

11.1.4. 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>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-kubernetes-config</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](#).

11.2. 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.

11.2.1. 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-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-config-archetype \
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004

```

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': : fuse79-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: fuse79-configmap
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y

```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse79-configmap** for the **artifactId** value. Accept the defaults for the remaining fields.

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

```

oc login -u developer -p developer
oc project openshift

```

3. At the command line, change to the directory of the new **fuse79-configmap** project and create the Secret object for this application.

```

cd fuse79-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](#).

4. Build and deploy the quickstart application. From the top level of the **fuse79-configmap** project, enter:

```

mvn oc:deploy -Popenshift

```

5. View the application log as follows.
 - 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**. Click **Status** to view the **Project Status** page. All the existing applications in the selected namespace (for example, openshift) are displayed.
 - c. Click **fuse79-configmap** to view the **Overview** information page for the quickstart.
 - d. In the left hand side panel, expand **Workloads**.
 - e. Click **Pods** and then click **fuse79-configmap-xxxx**. The pod details for the application are displayed.
 - f. Click on the **Logs** tab to view the application logs.

- 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:

```
5:44:57.377 [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
```

- Before you can update the configuration of the `fuse79-configmap` application using a ConfigMap object, you must give the `fuse79-configmap` application permission to view data from the OpenShift ApiServer. Enter the following command to give the `view` permission to the `fuse79-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`.

- 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:

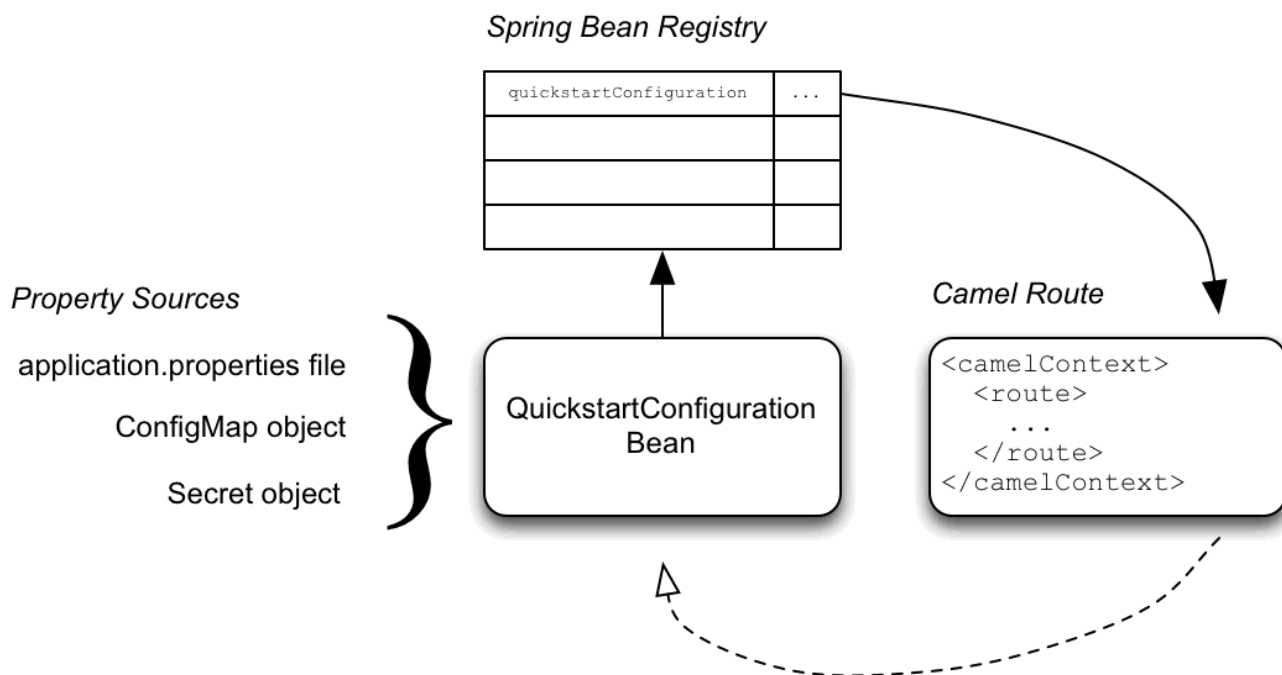
```
16:25:24.121 [Camel (camel) thread #0 - timer://order] INFO generate-order-route -
Generating message message-9, sending to the recipient list
16:25:24.124 [Camel (camel) thread #0 - timer://order] INFO target-route-queue - ---->
message-9 pushed to an async queue (simulation)
16:25:24.125 [Camel (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
```

11.2.2. 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 **fuse79-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.

11.2.3. 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: bXl1c2VyCg== 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/jkube/` 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.

11.2.4. 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](#).

11.3. 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](#)

11.3.1. 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
```

11.3.2. 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
```

11.3.3. 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
```

11.3.4. 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)
```

11.4. 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**.

11.4.1. 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=
```

11.4.2. 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
```

11.4.3. 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.

11.5. 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.

11.5.1. 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**.

11.5.2. 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.

11.5.3. 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.

11.5.4. 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.

11.5.5. 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 12. DEVELOPING AN APPLICATION FOR THE KARAF IMAGE

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

12.1. 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-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=karaf-camel-log-archetype \
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004
```

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': : fuse79-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: fuse79-karaf-camel-log
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

When prompted, enter **org.example.fis** for the **groupId** value and **fuse79-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 **fuse79-karaf-camel-log** subdirectory.
5. You are now ready to build and deploy the **fuse79-karaf-camel-log** project. Assuming you are still logged into OpenShift, change to the directory of the **fuse79-karaf-camel-log** project, and then build and deploy the project, as follows.

```
cd fuse79-karaf-camel-log
mvn oc:deploy -Popenshift
```

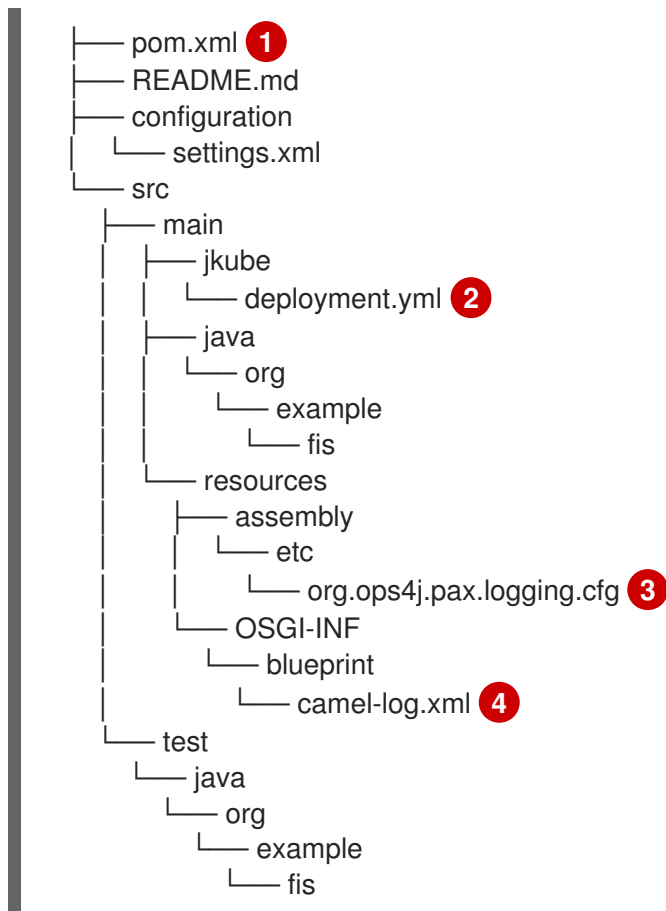


NOTE

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

12.2. 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/jkube/deployment.yml: Provides additional configuration that is merged with the default OpenShift configuration file generated by the openshift-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.

12.3. KARAF ARCHETYPE CATALOG

The Karaf archetype catalog includes the following examples.

Table 12.1. 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.

12.4. 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.

12.4.1. 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.

12.4.2. 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.

12.4.3. 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
```

12.4.4. 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}
```

12.4.5. 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)

12.4.6. 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

12.5. ADDING FABRIC8 KARAF CONFIG ADMIN SUPPORT

12.5.1. 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>
```

12.5.2. 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

```

12.5.3. 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
```

12.5.4. 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"/>

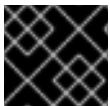
  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="userName" value="${amq.usr}"/>
    <property name="password" value="${amq.pwd}"/>
    <property name="brokerURL" value="${amq.url}"/>
  </bean>
</blueprint>

```

12.5.5. 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.

12.6. 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>
```

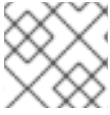


IMPORTANT

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

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

12.7.1. Configuring health checks

By default, the **fabric8-karaf-checks** endpoints are registered into the built-in HTTP server engine (Undertow) running on port **8181**. To avoid the health and readiness check requests being blocked by other long running HTTP processes in the container, the endpoints can be registered into a separate Undertow container.

These checks can be configured in the **etc/io.fabric8.checks.cfg** file by setting the following properties:

- **httpPort**: If this property is specified and is a valid port number, the **readiness-check** and **health-check** endpoints will be registered into a separate instance of Undertow server
- **readinessCheckPath** and **healthCheckPath** properties allow you to configure the actual URIs that can be used for readiness and health checks. By default these are the same as previous values.

**NOTE**

These properties may be changed after starting Fuse-Karaf, but may also be specified in **etc/io.fabric8.checks.cfg** file being part of custom Karaf distro, which is used by customers who want to have **fabric8-karaf-checks** feature running out of the box.

The following example illustrates the configuration of the health and readiness properties in the **etc/io.fabric8.checks.cfg** file:

Example

```
httpPort = 8182
readinessCheckPath = /readiness-check
healthCheckPath = /health-check
```

12.8. 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 getFailingHealthChecks();
        }

        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 13. 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](#).

13.1. 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.

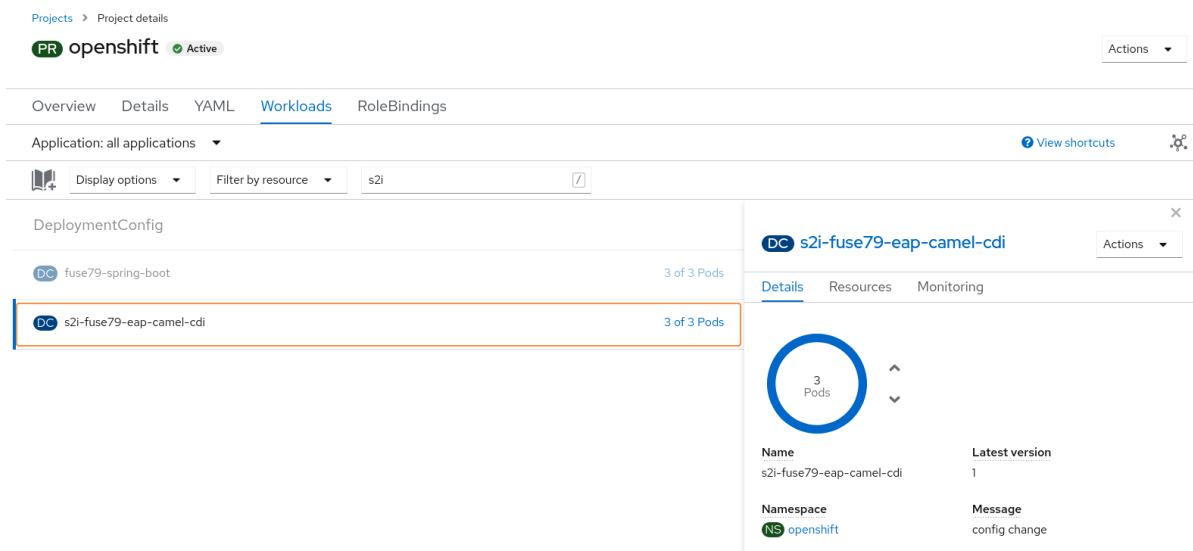
```
oc get template -n openshift
```

3. Enter the following command to create the resources required for running the **Red Hat Fuse 7.9 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.

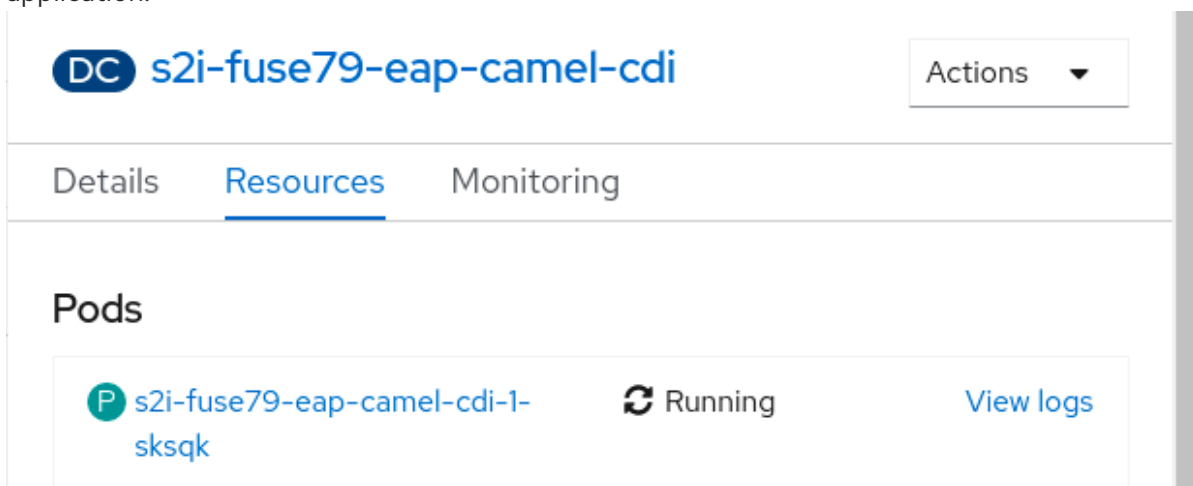
```
oc new-app s2i-fuse7-eap-camel-cdi

--> Creating resources ...
service "s2i-fuse7-eap-camel-cdi" created
service "s2i-fuse7-eap-camel-cdi-ping" created
route.route.openshift.io "s2i-fuse7-eap-camel-cdi" created
imagestream.image.openshift.io "s2i-fuse7-eap-camel-cdi" created
buildconfig.build.openshift.io "s2i-fuse7-eap-camel-cdi" created
deploymentconfig.apps.openshift.io "s2i-fuse7-eap-camel-cdi" created
--> Success
Access your application via route 's2i-fuse7-eap-camel-cdi-OPENSHIFT_IP_ADDR'
Build scheduled, use 'oc logs -f bc/s2i-fuse7-eap-camel-cdi' 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-fuse7-eap-camel-cdi** to view the **Overview** information page for the quickstart.



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



The link has the form http://s2i-fuse7-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-fuse7-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,

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



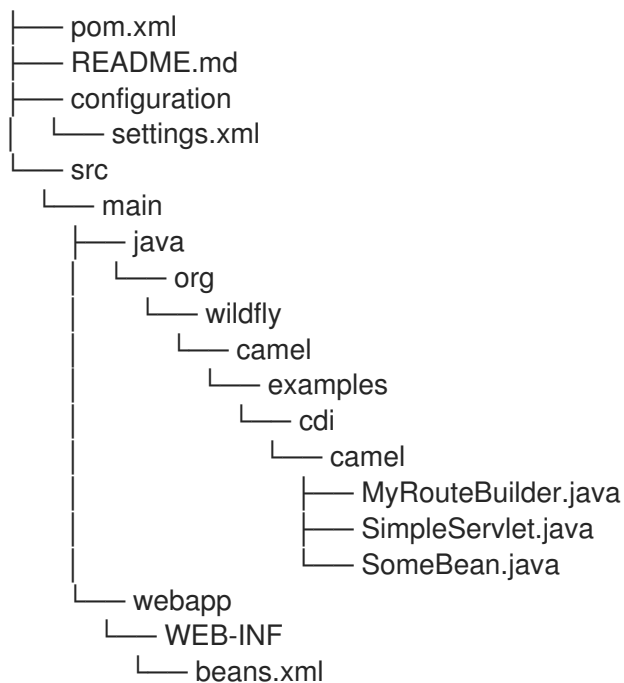
- b. Click the  icon next to Desired Count. The **Edit Count** window is displayed.
- c. Use the down arrow to scale down to zero to stop the pod.

13.2. STRUCTURE OF THE JBOSS EAP APPLICATION

You can find the source code for the Red Hat Fuse 7.9 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.

13.3. JBOSS EAP QUICKSTART TEMPLATES

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

Table 13.1. JBoss EAP S2I templates

Name	Description
------	-------------

Name	Description
JBoss Fuse 7.9 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 7.9 Camel CDI with EAP (eap-camel-cdi-template)	Demonstrates using the camel-cdi component to integrate CDI beans with Camel routes.
Red Hat Fuse 7.9 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 7.9 CXF JAX-WS with EAP (eap-camel-cxf-jaxws-template)	Demonstrates using the camel-cxf component to produce and consume JAX-WS web services.

CHAPTER 14. 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.

14.1. 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.

14.2. 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.

14.3. 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
```

14.4. 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
```

14.5. 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/jkube/deployment.yml` in a Fuse on OpenShift application and create OpenShift resources using command:

```
mvn oc: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 15. PATCHING FUSE ON OPENS SHIFT

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.

15.1. 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.

15.2. PATCHING THE FUSE ON OPENS SHIFT 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 you can download the updated images from registry.redhat.io. Fuse on OpenShift provides the following image streams (identified by their OpenShift *image stream name*):

- **fuse-java-openshift-rhel8**
- **fuse-java-openshift-jdk11-rhel8**
- **fuse-karaf-openshift-rhel8**
- **fuse-eap-openshift-jdk8-rhel7**
- **fuse-eap-openshift-jdk11-rhel8**

- **fuse-console-rhel8**
- **fuse-apicurito-generator-rhel8**
- **fuse-apicurito-rhel8**

Procedure

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

```
$ oc get is -n openshift
NAME                                DOCKER REPO                                TAGS
UPDATED
fuse-console-rhel8                  172.30.1.1:5000/openshift/fuse7/fuse-console-rhel8
1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9 About an hour ago
fuse7-eap-openshift-jdk8-rhel7      172.30.1.1:5000/openshift/fuse7/fuse-eap-openshift-
jdk8-rhel7 1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9 About an hour ago
fuse7-eap-openshift-jdk11-rhel8     172.30.1.1:5000/openshift/fuse7/fuse-eap-openshift-
jdk11-rhel8 1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9 About an hour ago
fuse7-java-openshift-rhel8          172.30.1.1:5000/openshift/fuse7/fuse-java-openshift-rhel8
1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9 About an hour ago
fuse7-java-openshift-jdk11-rhel8    172.30.1.1:5000/openshift/fuse7/fuse-java-openshift-
jdk11-rhel8 1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9 About an hour ago
fuse7-karaf-openshift-rhel8        172.30.1.1:5000/openshift/fuse7/fuse-karaf-openshift-rhel8
1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9 About an hour ago
fuse-apicurito-generator-rhel8      172.30.1.1:5000/openshift/fuse7/fuse-apicurito-generator-
rhel8 1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9 About an hour ago
apicurito-ui-rhel8                 172.30.1.1:5000/openshift/fuse7/apicurito-ui-rhel8
1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9 About an hour ago
```

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

```
oc import-image -n openshift fuse7/fuse7-java-openshift-rhel8:1.9
oc import-image -n openshift fuse7/fuse7-java-openshift-jdk11-rhel8:1.9
oc import-image -n openshift fuse7/fuse7-karaf-openshift-rhel8:1.9
oc import-image -n openshift fuse7/fuse7-eap-openshift-jdk8-rhel7:1.9
oc import-image -n openshift fuse7/fuse7-eap-openshift-jdk11-rhel8:1.9
oc import-image -n openshift fuse7/fuse7-console-rhel8:1.9
oc import-image -n openshift fuse7/apicurito-ui-rhel8:1.9
oc import-image -n openshift fuse7/fuse-apicurito-generator-rhel8:1.9
```



NOTE

The version tags in the image stream have the form **1.9-<BUILDNUMBER>**. When you specify the tag as **1.9**, you will get the latest build in the **1.9** 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 [Triggering and modifying builds](#) in the Builds OpenShift Container Platform documentation_.

15.3. 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

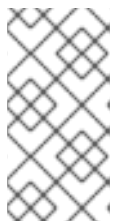
1. You need administrator privileges to update the Fuse on OpenShift templates. Log in to the OpenShift Server as an administrator, as follows:

```
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-2.1.0.fuse-sb2-790047-redhat-00005
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/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.

15.4. 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.

15.4.1. 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>7.9.0.fuse-sb2-790065-redhat-00001</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>openshift-maven-plugin</artifactId>
            ...
            <version>${fuse.version}</version>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>

```

```

</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 **openshift-maven-plugin** plugin and the **spring-boot-maven-plugin** plugin.

15.4.2. 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>7.9.0.fuse-sb2-790065-redhat-00001</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>openshift-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 **openshift-maven-plugin** plugin and the **karaf-maven-plugin** plugin.

15.4.3. 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>7.9.0.fuse-sb2-790065-redhat-00001</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.

15.5. AVAILABLE BOM VERSIONS

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

Table 15.1. Red Hat Fuse Releases and Corresponding New-Style BOM Version

Red Hat Fuse Release	org.jboss.redhat-fuse BOM Version
Red Hat Fuse 7.9 GA	7.9.0.fuse-sb2-790065-redhat-00001
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](#)

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>7.9.0.fuse-sb2-790065-redhat-00001</fuse.bom.version>

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

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

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

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

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

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

APPENDIX B. USING KARAF MAVEN PLUGIN

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

B.1. MAVEN DEPENDENCIES

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

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

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. OPENSIFT MAVEN PLUGIN

The OpenShift Maven plugin is used for building and deploying Java applications for OpenShift. It brings your Java applications on to OpenShift. It provides a tight integration into maven and benefits from the build configuration already provided. It focuses on three tasks:

- Building S2I images
- Creating OpenShift resources
- Deploy application on OpenShift

C.1. ABOUT OPENSIFT MAVEN PLUGIN

OpenShift Maven plugin has following features:

- Dealing with S2I images and hence inherits its flexible and powerful configuration.
- Supports both OpenShift descriptors
- OpenShift Docker builds with a binary source (as an alternative to a direct image build against a Docker daemon)
- Multiple configuration styles:
 - Zero Configuration for a quick ramp-up where opinionated defaults will be pre-selected.
 - Inline Configuration within the plugin configuration in an XML syntax.
 - External Configuration templates of the real deployment descriptors which are enriched by the plugin.
- Flexible customization:
 - Generators analyze the Maven build and generated automatic Docker image configurations for certain systems (spring-boot, plain java, karaf)
 - Enrichers extend the OpenShift resource descriptors by extra information like SCM labels and can add default objects like Services.
 - Generators and Enrichers can be individually configured and combined into profiles.

C.2. BUILDING IMAGES

The **oc:build** goal is used for creating Docker-formatted images containing an application. These then can be deployed later on Kubernetes or OpenShift. This plugin uses the assembly descriptor format from the **maven-assembly-plugin** to specify the content which will be added to the image. These images can then be pushed to public or private Docker registries with **oc:push**. The **oc:watch** goal allows for you to react to the code changes to automatically recreate images or copy new artifacts into running containers.

C.3. KUBERNETES AND OPENSIFT RESOURCES

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

C.4. INSTALLING OPENSIFT MAVEN PLUGIN

This plugin is available under the Maven central repository and can be connected to pre- and post-integration phases as shown below. By default, Maven will only search for plugins in the `org.apache.maven.plugins` and `org.codehaus.mojo` packages. In order to resolve the provider for the JKube plugin goals, edit the `~/.m2/settings.xml` file and add the **org.eclipse.jkube** namespace so the `<pluginGroups>` configuration.

Procedure

- To connect the OpenShift Maven plugin to pre- and post-integration phases, add the following to `~/.m2/settings.xml` file.

```
<settings>
  ...

  <pluginGroups>
    <pluginGroup>org.jboss.redhat-fuse</pluginGroup>
  </pluginGroups>

  ...
</settings>

<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>openshift-maven-plugin</artifactId>
  <version>${fuse.version}</version>

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

```

...
<build>
...
</build>
</image>
...
</images>
</configuration>

<!-- Connect oc:resource, oc:build and oc:helm to lifecycle phases -->
<executions>
<execution>
<id>jkube</id>
<goals>
<goal>resource</goal>
<goal>build</goal>
<goal>helm</goal>
</goals>
</execution>
</executions>
</plugin>

```

C.5. UNDERSTANDING OPENSIFT MAVEN PLUGIN BUILD GOALS

Build goals are used to create and manage the Kubernetes and OpenShift build artifacts like Docker-formatted images or S2I builds.

Table C.1. OpenShift Maven plugin build goals

Goal	Description
oc:resource	Create Kubernetes or OpenShift resource descriptors. Generated resources are in target/classes/META-INF/jkube/openshift directory.
oc:build	Build images.
oc:push	Push images to a registry. The registry to push is by default docker.io but can be specified as part of the images's name.
oc:apply	Apply resources to a running cluster. This goal is similar to oc:deploy but does not perform the full deployment cycle.

C.6. UNDERSTANDING OPENSIFT MAVEN PLUGIN DEVELOPMENT GOALS

Development goals are used in deploying resource descriptors to the development cluster. Also, helps you to manage the lifecycle of the development cluster.

Table C.2. OpenShift Maven plugin development goals

Goal	Description
oc:deploy	Deploy resources descriptors to a cluster after creating them and building the app. Same as oc:apply except that it runs in the background.
oc:undeploy	Undeploy and remove resources descriptors from a cluster.
oc:log	Show the logs of the running application.
oc:debug	Enable remote debugging.
oc:watch	Watch for file changes and perform rebuilds and redeployments.

APPENDIX D. FABRIC8 MAVEN PLUGIN



NOTE

Fabric8 Maven plugin is now **deprecated**. Use [OpenShift Maven plugin](#) to build and deploy your applications.

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.

D.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.

D.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>.

D.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>
```

D.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.

D.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 D.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 D.2. Development Goals

Goal	Description
<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.

D.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>

```

D.4.3. Resource validation configuration

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

Table D.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>.

D.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 D.5.3, "Spring Boot"](#)
- [Section D.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.

D.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.

D.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 **openshift-maven-plugin** is different from the Red Hat productized version (for example, in the community version, the default mode is **docker**).

D.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:1.9
fuse7/fuse-java-openshift:1.9
fuse7/fuse-karaf-openshift:1.9
```




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.

D.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:1.9
fuse7/fuse-java-openshift:1.9
fuse7/fuse-karaf-openshift:1.9
```

D.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>
```

D.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>
```

D.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>
```

D.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 D.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**.

D.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 D.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 E. 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.

E.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.

E.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>

```

E.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]
[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.

E.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 E.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).

E.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 F. CUSTOMIZING JVM ENVIRONMENT VARIABLES

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

F.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.

F.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:1.6 <target image name>
docker run <target image name>
```

F.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

F.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`
```

F.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.

F.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.

F.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 G. 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.

G.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

G.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.

G.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
```

G.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