Red Hat JBoss Fuse 6.2.1 Fuse Integration Services 1.0 for OpenShift

---

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

Red Hat JBoss Fuse Documentation Team

Red Hat JBoss Fuse 6.2.1 Fuse Integration Services 1.0 for OpenShift

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

Legal Notice

Abstract

Guide to using Fuse Integration Services 1.0 for OpenShift

Table of Contents

CHAPTER 1. INTRODUCTION

1.1. WHAT IS JBOSS FUSE INTEGRATION SERVICES?

Red Hat JBoss Fuse Integration Services provides a set of tools and containerized xPaaS images that enable development, deployment, and management of integration microservices within OpenShift.

Important

There are significant differences in supported configurations and functionality in Fuse Integration Services compared to the standalone JBoss Fuse product.

CHAPTER 2. BEFORE YOU BEGIN

2.1. COMPARISON: FUSE AND FUSE INTEGRATION SERVICES

There are several major functionality differences:

- Fuse Management Console is not included as Fuse administration views have been integrated directly within the OpenShift Web Console.
- An application deployment with Fuse Integration Services consists of an application and all required runtime components packaged inside a Docker image. Applications are not deployed to a runtime as with Fuse, the application image itself is a complete runtime environment deployed and managed through OpenShift.
- Patching in an OpenShift environment is different from standalone Fuse since 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.
- Messaging services are created and managed using the A-MQ for OpenShift image and not included directly within Fuse. Fuse Integration Services 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.

Additional details on technical differences and support scope are documented in an associated KCS article.

2.2. VERSION COMPATIBILITY AND SUPPORT

See the xPaaS part of the OpenShift and Atomic Platform Tested Integrations page for details about OpenShift image version compatibility.

2.3. INITIAL SETUP

The instructions in this guide follow on from and assume an OpenShift instance similar to that created in the OpenShift Primer.

CHAPTER 3. GET STARTED

You can start using Fuse Integration Services by creating an application and deploying it to OpenShift using one of the following application development workflows:

- Fabric8 Maven Workflow
- OpenShift Source-to-Image (S2I) Workflow

Both workflows begin with creating a new project from a Maven archetype.

3.1. MAVEN ARCHETYPES CATALOG

The Maven Archetype catalog includes the following examples:

cdi-camel-http-archetype	Creates a new Camel route using CDI in a standalone Java Container calling the remote camel-servlet quickstart
cdi-cxf-archetype	Creates a new CXF JAX-RS using CDI running in a standalone Java Container
cdi-camel-archetype	Creates a new Camel route using CDI in a standalone Java Container
cdi-camel-jetty-archetype	Creates a new Camel route using CDI in a standalone Java Container using Jetty as HTTP server
java-simple-mainclass-archetype	Creates a new Simple standalone Java Container (main class)
java-camel-spring-archetype	Creates a new Camel route using Spring XML in a standalone Java container
karaf-cxf-rest-archetype	Creates a new RESTful WebService Example using JAX-RS
karaf-camel-rest-sql-archetype	Creates a new Camel Example using Rest DSL with SQL Database

karaf-camel-log-archetype	Creates a new Camel Log Example

Begin by selecting the archetype which matches the type of application you would like to create.

3.2. CREATE AN APPLICATION FROM THE MAVEN ARCHETYPE CATALOG

You must configure the Maven repositories, which hold the archetypes and artifacts you may need, before creating a sample project:

- JBoss Fuse repository: **https://repo.fusesource.com/nexus/content/groups/public/**
- RedHat GA repository: **https://maven.repository.redhat.com/ga**

Add above repositories to the dependency repositories section as well as plugin repositories section of your **.m2/settings.xml** file. For more information on adding maven repositories, see Preparing to use Maven section.

Use the maven archetype catalog to create a sample project with the required resources. The command to create a sample project is:

```
$ mvn archetype:generate \
  -
DarchetypeCatalog=https://repo.fusesource.com/nexus/content/groups/publ
ic/archetype-catalog.xml \
  -DarchetypeGroupId=io.fabric8.archetypes \
  -DarchetypeVersion=2.2.0.redhat-079 \
  -DarchetypeArtifactId=<archetype-name>
```

Note

Replace <archetype-name> with the name of the archetype that you want to use. For example, **karaf-camel-log-archetype** creates a new Camel log example.

This will create a maven project with all required dependencies. Maven properties and plug-ins that are used to create Docker images are added to the ***pom.xml*** file.

3.3. FABRIC8 MAVEN WORKFLOW

Creates a new project based off a Maven application template created through Archetype catalog. This catalog provides examples of Java and Karaf projects and supports S2I and Maven deployment workflows.

1. Set the following environment variables to communicate with OpenShift and a Docker daemon:

DOCKER_HOST	Specifies the connection to a Docker daemon used to build an application Docker image	**tcp://10.1.2.2:2375**
KUBERNETES_MASTER	Specifies the URL for contacting the OpenShift API server	**https://10.1.2.2:8443**
KUBERNETES_DOMAIN	Domain used for creating routes. Your OpenShift API server must be mapped to all hosts of this domain.	**openshift.dev**

2. Login to OpenShift using CLI and select the project to which to deploy.

```
$ oc login

$ oc project <projectname>
```

3. Create a sample project as described in Create an Application from the Maven Archetype Catalog.

4. Build and push the project to OpenShift. You can use following maven goals for building and pushing docker images.

docker:build	Builds the docker image for your maven project.
docker:push	Pushes the locally built docker image to the global or a local docker registry. This step is optional when developing on a single node OpenShift cluster.
fabric8:json	Generates kubernetes json file for your maven project. This goal is bound to the **package** phase and doesn't need to be called explicitly when running **mvn install**
fabric8:apply	Applies the kubernetes json file to the current Kubernetes environment and namespace.

There are few pre-configured maven profiles that you can use to build the project. These profiles are combinations of above maven goals that simplify the build process.

mvn -Pf8-build	Comprises of **`clean`**, **`install`**, **`docker:build`**, and **`fabric8:json`**. This will build dockerfile and JSON template for a project.
mvn -Pf8-local-deploy	Comprises of **`clean`**, **`install`**, **`docker:build`**, **`fabric8:json`**, and **`fabric8:apply`**. This will create docker and JSON templates and then apply them to OpenShift.
mvn -Pf8-deploy	Comprises of **`clean`**, **`docker:build`**, **`fabric8:json`**, **`docker:push`**, and **`fabric8:apply`**. This will create docker and JSON templates, push them to docker registry and apply to OpenShift.

a. To deploy the application use **`-Pf8-deploy`** and **`fabric8:json`** goals which create docker and JSON templates, push them to docker registry and apply to OpenShift.

```
$ mvn -Pf8-deploy -Ddocker.registry=registry.openshift.dev
-Ddocker.username=$(oc whoami) -Ddocker.password=$(oc
whoami -t) -Dfabric8.dockerUser=$(oc project -q)/

$ mvn fabric8:json fabric8:apply -Dfabric8.dockerUser=$(oc
get svc docker-registry -n default -o 'jsonpath=
{.spec.clusterIP}:{.spec.ports[0].port}')/$(oc project -q)/
```

Note

The **`docker-registry`** pod and **`default`** namespace may differ from environment to environment. You may or may not have access to pull information on this object (depending on cluster configuration). Hence, this command may not work in some cases and you may have to enter the IP:PORT information manually.

b. When you are using OpenShift cluster, you may want to push your application to an external registry. You can use **`-Pf8-deploy`** goal to build your application and deploy it to external registry.

```
$ mvn -Pf8-deploy -Ddocker.registry=registry.openshift.dev
-Ddocker.username=my-registry-user -Ddocker.password=my-
registry-password -
Dfabric8.dockerUser=registry.openshift.dev/fabric8/
```

For more information about exposing the registry refer Exposing the registry section of OpenShift Enterpise Installation and Configuration guide

Note

In case of local development environment, it is not required to use **`docker:push`**. You can use **`-Pf8-local-deploy`** goal which creates docker and JSON templates and then apply them to OpenShift.

1. Login to OpenShift Web Console. A pod is created for the newly created application. You can view the status of this pod, deployments and services that the application is creating.

3.3.1. Authenticating Against a Registry

For multi node OpenShift setups, the image created must be pushed to the OpenShift registry. This registry must be reachable from the outside through a route. Authentication against this registry reuses the OpenShift authentication with **`oc login`**. Assuming that your OpenShift registry is exposed as **`registry.openshift.dev:80`**, the project image can be deployed to the registry with following command:

```
$ mvn docker:push -Ddocker.registry=registry.openshift.dev:80 \
                  -Ddocker.username=$(oc whoami) \
                  -Ddocker.password=$(oc whoami -t)
```

To push changes to the registry, the OpenShift project must exist and the users of Docker image must be connected to the OpenShift project. All the examples uses the property **`fabric8.dockerUser`** as Docker image user which has **`fabric8/`** as default (note the trailing slash). When this user is used unaltered an OpenShift project 'fabric8' must exist. This can be created with 'oc new-project fabric8'.

3.3.2. Plug-in Configuration

Plug-ins **`docker-maven-plugin`** and **`fabric8-maven-plugin`** are responsible for creating Docker images and OpenShift API objects which can be configured flexibly. The examples from the archetypes introduces some extra properties which can be changed when running Maven:

docker.registry	Registry to use for **`docker:push`** and **`-Pf8-deploy`**
docker.username	Username for authentication against the registry

docker.password	Password for authentication against the registry
docker.from	Base image for the application Docker image
fabric8.dockerUser	User used in the image's name as user part. It must contain a `/` as trailing part. The default value is **`fabric8/`**.
docker.image	The final Docker image name. Default value is **`${fabric8.dockerUser}${project.artifactId}:${project.version}`**

3.4. OPENSHIFT SOURCE-TO-IMAGE (S2I) WORKFLOW

Applications are created through OpenShift Admin Console and CLI using application templates. If you have a JSON or YAML file that defines a template, you can upload the template to the project using the CLI. This saves the template to the project for repeated use by users with appropriate access to that project. You can add the remote Git repository location to the template using template parameters. This allows you to pull the application source from remote repository and built using source-to-image (S2I) method.

JBoss Fuse Integration Services application templates depend on S2I builder **`ImageStreams`**, which MUST be created ONCE. The OpenShift installer creates them automatically. For users existing OpenShift setups, it can be achieved with the following command:

```
$ oc create -n openshift -f /usr/share/openshift/examples/xpaas-
streams/fis-image-streams.json
```

The **`ImageStreams`** may be created in a namespace other than **openshift** by changing it in the command and corresponding template parameter **`IMAGE_STREAM_NAMESPACE`** when creating applications.

3.4.1. Create an Application Using Templates

1. Create an application template using command **`mvn archetype:generate`**. To create an application, upload the template to your current project's template library with the following command:

   ```
   $ oc create -f quickstart-template.json -n <project>
   ```

   The template is now available for selection using the web console or the CLI.

2. Login to OpenShift Web Console. In the desired project, click **Add to Project** to create the objects from an uploaded template.

3. Select the template from the list of templates in your project or from the global template library.

4. Edit template parameters and then click **Create**. For example, template parameters for a camel-spring quickstart are:

Parameter	Description	Default
APP_NAME	Application Name	Artifact name of the project
GIT_REPO	Git repository, required	
GIT_REF	Git ref to build	`master`
SERVICE_NAME	Exposed Service name	
BUILDER_VERSION	Builder version	1.0
APP_VERSION	Application version	Maven project version
MAVEN_ARGS	Arguments passed to mvn in the build	`package -DskipTests -e`
MAVEN_ARGS_APPEND	Extra arguments passed to mvn, e.g. for multi-module builds use `-pl groupId:module-artifactId -am`	
ARTIFACT_DIR	Maven build directory	`target/`
IMAGE_STREAM_NAMESPACE	Namespace in which the JBoss Fuse ImageStreams are installed.	
BUILD_SECRET	generated if empty. The secret needed to trigger a build.	

5. After successful creation of the application, you can view the status of application by clicking **Pods** tab or by running the following command:

```
$ oc get pods
```

For more information, see Application Templates.

3.5. DEVELOPING APPLICATIONS

3.5.1. Injecting Kubernetes Services into Applications

You can inject Kubernetes services into applications by labeling the pods and use those labels to select the required pods to provide a logical service. These labels are simple key, value pairs.

3.5.1.1. CDI Injection

Fabric8 provides a CDI extension that you can use to inject Kubernetes resources into your applications. To use the CDI extension, first add the dependency to the project's ***pom.xml*** file.

```
<dependency>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-cdi</artifactId>
  <version>{$fabric8.version}</version>
</dependency>
```

Next step is to identify the field that requires the service and then inject the service by adding a **`@ServiceName`** annotation to it. For example,

```
@Inject
@ServiceName("my-service")
private String service.
```

The **`@PortName`** annotation is used to select a specific port by name when multiple ports are defined for a service.

3.5.1.2. Using Environment Variables as Properties

You can use to access a service by using environment variables to expose the fixed IP address and port. These are, **`SERVICE_HOST`** and **`SERVICE_PORT`**. **`SERVICE_HOST`** is the host (IP) address of the service and **`SERVICE_PORT`** is the port of the service.