fabric8io/fabric8-maven-plugin

Roland Huß, James Strachan

Version 3.5.41, 2018-09-13

fabric8-maven-plugin

1. Introduction . . . 2
   1.1. Building Images . . . 2
   1.2. Kubernetes and OpenShift Resources . . . 2
   1.3. Configuration . . . 2
   1.4. Examples . . . 4
      1.4.1. Zero-Config . . . 4
      1.4.2. XML Configuration . . . 7
      1.4.3. Resource Fragments . . . 9
      1.4.4. Docker Compose . . . 10
2. Compatibility with OpenShift and Kubernetes . . . 12
   2.1. OpenShift Compatibility . . . 12
   2.2. Kubernetes Compatibility . . . 12
3. Installation . . . 13
4. Goals Overview . . . 14
5. Build Goals . . . 15
   5.1. **fabric8:resource** . . . 15
      5.1.1. Labels and Annotations . . . 15
      5.1.2. Secrets . . . 17
      5.1.3. Resource Validation . . . 18
      5.1.4. Route Generation . . . 19
      5.1.5. Disable Automatic Deployments . . . 20
   5.2. **fabric8:build** . . . 20
      5.2.1. Kubernetes Build . . . 20
      5.2.2. OpenShift Build . . . 20
      5.2.3. Configuration . . . 21
      5.2.4. Image Configuration . . . 24
      5.2.5. Build Configuration . . . 26
      5.2.6. Assembly . . . 31
      5.2.7. Environment and Labels . . . 35
      5.2.8. Startup Arguments . . . 36
      5.2.9. Build Args . . . 37
   5.3. **fabric8:push** . . . 38
   5.4. **fabric8:apply** . . . 38
   5.5. **fabric8:resource-apply** . . . 39
   5.6. **fabric8:helm** . . . 40
6. Development Goals . . . 42
   6.1. **fabric8:deploy** . . . 42
   6.2. **fabric8:undeploy** . . . 43

6.3. **fabric8:log** . . . 43
6.4. **fabric8:debug** . . . 44
6.4.1. Speeding up debugging . . . 44
6.4.2. Debugging with suspension . . . 45
6.5. **fabric8:watch** . . . 46
6.5.1. Spring Boot . . . 46
6.5.2. Docker Image . . . 47
7. Generators . . . 48
7.1. Default Generators . . . 50
7.1.1. Java Applications . . . 51
7.1.2. Spring Boot . . . 52
7.1.3. Wildfly Swarm . . . 53
7.1.4. Thorntail v2 . . . 53
7.1.5. Vert.x . . . 53
7.1.6. Karaf . . . 54
7.1.7. Web Applications . . . 55
7.2. Generator API . . . 56
8. Enrichers . . . 57
8.1. Default Enrichers . . . 57
8.1.1. Standard Enrichers . . . 59
8.1.2. Fabric8 Enrichers . . . 64
8.1.3. OpenShift.io Enrichers . . . 78
8.2. Enricher API . . . 80
9. Profiles . . . 81
9.1. Generator and Enricher definitions . . . 82
9.2. Lookup order . . . 82
9.3. Using Profiles . . . 83
9.4. Predefined Profiles . . . 84
9.5. Extending Profiles . . . 85
10. Access configuration . . . 87
10.1. Docker Access . . . 87
10.2. OpenShift and Kubernetes Access . . . 87
11. Registry handling . . . 88
12. Authentication . . . 90
12.1. Pull vs. Push Authentication . . . 91
12.2. OpenShift Authentication . . . 92
12.3. Password encryption . . . 92
12.4. Extended Authentication . . . 93
13. Migration from version 2 . . . 94
14. FAQ . . . 95
14.1. General questions . . . 95

14.1.1. How do I define an environment variable? 95
14.1.2. How do I define a system property? 95
14.1.3. How do I mount a config file from a ConfigMap? 95
14.1.4. How do I use a Persistent Volume? 96

© 2016 The original authors.

Chapter 1. Introduction

The **fabric8-maven-plugin** (f8-m-p) brings your Java applications on to Kubernetes and OpenShift. It provides a tight integration into Maven and benefits from the build configuration already provided. This plugin focus on two tasks: *Building Docker images* and *creating Kubernetes and OpenShift resource descriptors*. It can be configured very flexibly and supports multiple configuration models for creating: A *Zero-Config* setup allows for a quick ramp-up with some opinionated defaults. For more advanced requirements, an *XML configuration* provides additional configuration options which can be added to the `pom.xml`. For the full power, in order to tune all facets of the creation, external *resource fragments* and *Dockerfiles* can be used. A docker compose configuration can be also used to bring up docker compose deployments on a Kubernetes/OpenShift cluster.

1.1. Building Images

The **fabric8:build** goal is for creating Docker images containing the actual application. These then can be deployed later on Kubernetes or OpenShift. It is easy to include build artifacts and their dependencies into 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. That images can then be pushed to public or private Docker registries with **fabric8:push**.

Depending on the operational mode, for building the actual image either a Docker daemon is used directly or an OpenShift Docker Build is performed.

A special **fabric8:watch** goal allows for reacting to code changes to automatically recreate images or copy new artifacts into running containers.

These image related features are inherited from the fabric8io/docker-maven-plugin which is part of this plugin.

1.2. Kubernetes and OpenShift Resources

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

Typically you only specify a small part of the real resource descriptors which will be enriched by this plugin with various extra information taken from the `pom.xml`. This drastically reduces boilerplate code for common scenarios.

1.3. Configuration

As mentioned already there are four levels of configuration:

- **Zero-Config** mode makes some very opinionated decisions based on what is present in the pom.xml like what base image to use or which ports to expose. This is great for starting up things and for keeping quickstart applications small and tidy.

- **XML plugin configuration** mode is similar to what docker-maven-plugin provides. This allows for type-safe configuration with IDE support, but only a subset of possible resource descriptor features is provided.
- **Kubernetes & OpenShift resource fragments** are user provided YAML files that can be *enriched* by the plugin. This allows expert users to use a plain configuration file with all their capabilities, but also to add project specific build information and avoid boilerplate code.
- **Docker Compose** can be used to bring up docker compose deployments on a Kubernetes/OpenShift cluster. This requires minimum to no knowledge of Kubernetes/OpenShift deployment process.

The following table gives an overview of the different models

Table 1. Configuration Models

Model	Docker Images	Resource Descriptors
Zero-Config	Generators are used to create Docker image configurations. Generators can detect certain aspects of the build (e.g. whether Spring Boot is used) and then choose some default like the base image, which ports to expose and the startup command. They can be configured, but offer only a few options.	Default Enrichers will create a default *Service* and *Deployment* (*DeploymentConfig* for OpenShift) when no other resource objects are provided. Depending on the image they can detect which port to expose in the service. As with Generators, Enrichers support a limited set of configuration options.
XML configuration	f8-m-p inherits the XML based configuration for building images from the docker-maven-plugin and provides the same functionality. It supports an assembly descriptor for specifying the content of the Docker image.	A subset of possible resource objects can be configured with a dedicated XML syntax. With a decent IDE you get autocompletion on most object and inline documentation for the available configuration elements. The provided configuration can be still enhanced by Enhancers which is useful for adding e.g. labels and annotations containing build or other information.
Resource Fragments and **Dockerfiles**	Similarly to the docker-maven-plugin, f8-m-p supports external Dockerfiles too, which are referenced from the plugin configuration.	Resource descriptors can be provided as external YAML files which specify a skeleton. This skeleton is then filled by Enrichers which add labels and more. Maven properties within these files are resolved to thier values. With this model you can use every Kubernetes / OpenShift resource object with all their flexibility, but still get the benefit of adding build information.

Model	Docker Images	Resource Descriptors
Docker Compose	All above methods can works with docker compose configuration.	The plugin converts docker compose descriptors into more detailed Kubernetes/OpenShift deployment resources. These resource descriptors or externally YAML files can be used to specify the application skeleton. This skeleton is then filled by Enrichers which add labels and more. Maven properties within these files are resolved to their values. With this model you can use every Kubernetes / OpenShift resource objects with all their flexibility, but still get the benefit of adding build information.

1.4. Examples

Let's have a look at some code. The following examples will demonstrate all three configurations variants:

1.4.1. Zero-Config

This minimal but full working example `pom.xml` shows how a simple spring boot application can be dockerized and prepared for Kubernetes and OpenShift. The full example can be found in directory samples/zero-config.

Example

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

  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-sample-zero-config</artifactId>
  <version>3.5.41</version>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId> ①
    <version>1.5.5.RELEASE</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId> ②
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId> ③
      </plugin>
      <plugin>
        <groupId>io.fabric8</groupId>
        <artifactId>fabric8-maven-plugin</artifactId> ④
        <version>3.5.41</version>
      </plugin>
    </plugins>
  </build>
</project>
```

① This minimalistic spring boot application uses the spring-boot parent POM for setting up dependencies and plugins

② The Spring Boot web starter dependency enables a simple embedded Tomcat for serving Spring MVC apps

③ The `spring-boot-maven-plugin` is responsible for repackaging the application into a fat jar, including all dependencies and the embedded Tomcat

④ The `fabric8-maven-plugin` enables the automatic generation of a Docker image and Kubernetes / OpenShift descriptors including this Spring application.

This setup make some opinionated decisions for you:

- As base image fabric8/java-jboss-openjdk8-jdk is chosen which enables Jolokia and jmx_exporter. It also comes with a sophisticated startup script.
- It will create a Kubernetes Deployment and a Service as resource objects
- It exports port 8080 as the application service port (and 8778 and 9779 for Jolokia and jmx_exporter access, respectively)

These choices can be influenced by configuration options as decribed in Spring Boot Generator.

To start the Docker image build, you simply run

```
mvn package fabric8:build
```

This will create the Docker image against a running Docker daemon (which must be accessible either via Unix Socket or with the URL set in `DOCKER_HOST`). Alternatively, when connected to an OpenShift cluster (or using the `openshift` mode explicitly), then a Docker build will be performed on OpenShift which at the end creates an ImageStream.

To deploy the resources to the cluster call

```
mvn fabric8:resource fabric8:deploy
```

By default a *Service* and a *Deployment* object pointing to the created Docker image is created. When running in OpenShift mode, a *Service* and *DeploymentConfig* which refers the *ImageStream* created with `fabric8:build` will be installed.

Of course you can bind all those fabric8-goals to execution phases as well, so that they are called along with standard lifecycle goals like `install`. For example, to bind the building of the Kubernetes resource files and the Docker images, add the following goals to the execution of the f-m-p:

Example for lifecycle bindings

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

  <!-- ... -->

  <executions>
    <execution>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

If you'd also like to automatically deploy to Kubernetes each time you do a `mvn install` you can add the `deploy` goal:

Example for lifecycle bindings with automatic deploys for mvn install

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

  <!-- ... -->

  <executions>
    <execution>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
        <goal>deploy</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

1.4.2. XML Configuration

XML based configuration is only partially implemented and is not recommended for use right now.

Although the Zero-config mode and its generators can be tweaked with options up to a certain degree, many cases require more flexibility. For such instances, an XML-based plugin configuration can be used, in a way similar to the XML configuration used by `docker-maven-plugin`.

The plugin configuration can be roughly divided into the following sections:

- Global configuration options are responsible for tuning the behaviour of plugin goals
- `<images>` defines which Docker images are used and configured. This section is similar to the image configuration of the `docker-maven-plugin`, except that `<run>` and `<external>` sub-elements are ignored)
- `<resource>` defines the resource descriptors for deploying on an OpenShift or Kuberneres cluster.
- `<generator>` configures generators which are responsible for creating images. Generators are used as an alternative to a dedicated `<images>` section.
- `<enricher>` configures various aspects of enrichers for creating or enhancing resource descriptors.

A working example can be found in the samples/xml-config directory. An extract of the plugin configuration is shown below:

Example for an XML configuration

```
<configuration>
  <images>  ①
    <image>
      <name>xml-config-demo:1.0.0</name>
      <!-- "alias" is used to correlate to the containers in the pod spec -->
      <alias>camel-app</alias>
      <build>
        <from>fabric8/java</from>
        <assembly>
          <basedir>/deployments</basedir>
          <descriptorRef>artifact-with-dependencies</descriptorRef>
        </assembly>
        <env>
          <JAVA_LIB_DIR>/deployments</JAVA_LIB_DIR>
          <JAVA_MAIN_CLASS>org.apache.camel.cdi.Main</JAVA_MAIN_CLASS>
        </env>
      </build>
    </image>
  </images>

  <resources> ②
    <labels> ③
      <all>
        <group>quickstarts</group>
      </all>
    </labels>

    <deployment> ④
      <name>${project.artifactId}</name>
      <replicas>1</replicas>

      <containers> ⑤
        <container>
          <alias>camel-app</alias> ⑥
          <ports>
            <port>8778</port>
          </ports>
          <mounts>
            <scratch>/var/scratch</scratch>
          </mounts>
        </container>
      </containers>

      <volumes> ⑦
        <volume>
          <name>scratch</name>
          <type>emptyDir</type>
        </volume>
      </volumes>
    </deployment>
```

```
    <services> ⑧
      <service>
        <name>camel-service</name>
        <headless>true</headless>
      </service>
    </services>
  </resources>
</configuration>
```

① Standard docker-maven-plugin configuration for building one single Docker image

② Kubernetes / OpenShift resources to create

③ Labels which should be applied globally to all resource objects

④ Definition of a Deployment to create

⑤ Containers to include in the deployment

⑥ An *alias* is used to correlate a container's image with the image definition in the `<images>` section where each image carry an alias. Can be omitted if only a single image is used

⑦ Volume definitions used in a Deployment's *ReplicaSet*

⑧ One or more Service definitions.

The XML resource configuration is based on plain Kubernetes resource objects. When targeting OpenShift, Kubernetes resource descriptors will be automatically converted to their OpenShift counterparts, e.g. a Kubernetes Deployment will be converted to an OpenShift DeploymentConfig.

1.4.3. Resource Fragments

The third configuration option is to use an external configuration in form of YAML resource descriptors which are located in the `src/main/fabric8` directory. Each resource get its own file, which contains a skeleton of a resource descriptor. The plugin will pick up the resource, enrich it and then combine all to a single `kubernetes.yml` and `openshift.yml` file. Within these descriptor files you are can freely use any Kubernetes feature.

Note: In order to support simultaneously both OpenShift and Kubernetes, there is currently no way to specify OpenShift-only features this way, though this might change in future releases.

Let's have a look at an example from samples/external-resources. This is a plain Spring Boot application, whose images are auto generated like in the Zero-Config case. The resource fragments are in `src/main/fabric8`.

Example fragment "deployment.yml"

```
spec:
  replicas: 1
  template:
    spec:
      volumes:
        - name: config
          gitRepo:
            repository: 'https://github.com/jstrachan/sample-springboot-config.git'
            revision: 667ee4db6bc842b127825351e5c9bae5a4fb2147
            directory: .
      containers:
        - volumeMounts:
            - name: config
              mountPath: /app/config
          env:
            - name: KUBERNETES_NAMESPACE
              valueFrom:
                fieldRef:
                  apiVersion: v1
                  fieldPath: metadata.namespace
      serviceAccount: ribbon
```

As you can see, there is no `metadata` section as would be expected for Kubernetes resources because it will be automatically added by the `fabric8-maven-plugin`. The object's `Kind`, if not given, is automatically derived from the filename. In this case, the `fabric8-maven-plugin` will create a `Deployment` because the file is called `deployment.yml`. Similar mappings between file names and resource type exist for each supported resource kind, the complete list of which (along with associated abbreviations) can be found at https://github.com/fabric8io/fabric8-maven-plugin/blob/master/core/src/main/java/io/fabric8/maven/core/util/kubernetes/KubernetesResourceUtil.java#L198-L244.

Additionally, if you name your fragment using a name prefix followed by a dash and the mapped file name, the plugin will automatically use that name for your resource. So, for example, if you name your deployment fragment `myapp-deployment.yml`, the plugin will name your resource `myapp`. In the absence of such provided name for your resource, a name will be automatically derived from your project's metadata (in particular, its `artifactId` as specified in your POM).

No image is also referenced in this example because the plugin also fills in the image details based on the configured image you are building with (either from a generator or from a dedicated image plugin configuration, as seen before).

1.4.4. Docker Compose

The fourth configuration option is to provide an external Docker Compose file. The following are some ways to specify docker-compose files.

1. Put the Docker Compose file into `src/main/fabric8-compose` directory in the project space (only

one file is supported at the moment).

2. Locate the Docker Compose file path using the plugin configuration, as shown in following `pom.xml` example.

Example fragment "pom.xml"

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

  <!-- ... -->

  <configuration>
    <composeFile>docker-compose.yaml</composeFile>
  </configuration>

  <executions>
    <execution>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

As you can see in `pom.xml`, the `composeFile` section under plugin `configuration` is used to provide the Docker Compose file path. It should be a relative path from the directory containing the `pom.xml` file.

If the execution configuration includes the `resource` goal (as shown in the `pom.xml` file above), the plugin will process the Docker Compose file and generate Kubernetes/OpenShift resource descriptors during the build. Resources can be also generated using the **fabric8:resource** goal directly. The Docker Compose option is implemented using kompose project.

A working example can be found in the samples/docker-compose

For building images there is also an alternative mode using external Dockerfiles, in addition to the XML based configuration. Refer to fabric8:build for details.

Enrichment of resource fragments can be fine-tuned by using profile sub-directories. For more details see Profiles.

Now that we have seen some examples for the various ways how this plugin can be used, the following sections will describe the plugin goals and extension points in detail.

Chapter 2. Compatibility with OpenShift and Kubernetes

2.1. OpenShift Compatibility

Table 2. OpenShift Comptatibility

FMP	OpenShift 3.9.0	OpenShift 3.7.0	OpenShift 3.6.0	OpenShift 3.5.0	OpenShift 1.4.1
FMP 3.5.38	✓	✓	✓	x	x
FMP 3.5.37	✓	✓	✓	x	x
FMP 3.5.36	✓	✓	✓	x	x
FMP 3.5.35	✓	✓	✓	x	x
FMP 3.5.34	✓	✓	✓	x	x
FMP 3.5.33	✓	✓	✓	x	x
FMP 3.5.32	✓	✓	✓	✓	✓

2.2. Kubernetes Compatibility

Table 3. Kubernetes Compatibility

FMP	Kubernetes 1.9.0	Kubernetes 1.8.0	Kubernetes 1.7.0	Kubernetes 1.6.0	Kubernetes 1.5.1	Kubernetes 1.4.0
FMP 3.5.38	✓	✓	✓	✓	✓	✓
FMP 3.5.37	✓	✓	✓	✓	✓	✓
FMP 3.5.36	✓	✓	✓	✓	✓	✓
FMP 3.5.35	✓	✓	✓	✓	✓	✓
FMP 3.5.34	✓	✓	✓	✓	✓	✓
FMP 3.5.33	✓	✓	✓	✓	✓	✓
FMP 3.5.32	✓	✓	✓	✓	✓	✓

Chapter 3. Installation

This plugin is available from Maven central and can be connected to pre- and post-integration phase as seen below. The configuration and available goals are described below.

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.41</version>

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

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

Chapter 4. Goals Overview

This plugin supports a rich set for providing a smooth Java developer experience. These goals can be categorized in multiple groups:

- Build goals are all about creating and managing Kubernetes and OpenShift build artifacts like Docker images or S2I builds.
- Development goals target help not only in deploying resource descriptors to the development cluster but also to manage the lifecycle of the development cluster as well.

Table 4. Build Goals

Goal	Description
fabric8:build	Build images
fabric8:push	Push images to a registry
fabric8:resource	Create Kubernetes or OpenShift resource descriptors
fabric8:apply	Apply resources to a running cluster
fabric8:resource-apply	Run `fabric8:resource` `fabric8:apply`

Table 5. Development Goals

Goal	Description
fabric8:deploy	Deploy resources decriptors to a cluster after creating them and building the app. Same as [fabric8:run] except that it runs in the backgorund.
fabric8:undeploy	Undeploy and remove resources decriptors from a cluster.
fabric8:watch	Watch for file changes and perform rebuilds and redeployments
fabric8:log	Show the logs of the running application
fabric8:debug	Enable remote debugging

Depending on whether the OpenShift or Kubernetes operational mode is used, the workflow and the performed actions differs :

Table 6. Workflows

Use Case	Kubernetes	OpenShift
Build	`fabric8:build` `fabric8:push` * Creates a image against an exposed Docker daemon (with a `docker.tar`) * Pushes the image to a registry which is then referenced from the configuration	`fabric8:build` * Creates or uses a `BuildConfig` * Creates or uses an `ImageStream` which can be referenced by the deployment descriptors in a `DeploymenConfig` * Starts an OpenShift build with a `docker.tar` as input
Deploy	`fabric8:deploy` * Applies a Kubernetes resource descriptor to cluster	`fabric8:deploy` * Applies an OpenShift resource descriptor to a cluster

Chapter 5. Build Goals

5.1. fabric8:resource

This is chapter is incomplete, but there is work in progress.

5.1.1. Labels and Annotations

Labels and annotations can be easily added to any resource object. This is best explained by an example.

Example for label and annotations

```
<plugin>
  ...
  <configuration>
    ...
    <resources>
      <labels> ①
        <all> ①
          <property> ②
            <name>organisation</name>
            <value>unesco</value>
          </property>
        </all>
        <service> ③
          <property>
            <name>database</name>
            <value>mysql</value>
          </property>
          <property>
            <name>persistent</name>
            <value>true</value>
          </property>
        </service>
        <replicaSet> ④
          ...
        </replicaSet>
        <pod> ⑤
          ...
        </pod>
        <deployment> ⑥
          ...
        </deployment>
      </labels>

      <annotations> ⑦
         ...
      </annotations>
    </resource>
  </configuration>
</plugin>
```

① `<labels>` section with `<resources>` contains labels which should be applied to objects of various kinds

② Within `<all>` labels which should be applied to **every** object can be specified

③ `<service>` labels are used to label services

④ `<replicaSet>` labels are for replica set and replication controller

⑤ `<pod>` holds labels for pod specifications in replication controller, replica sets and deployments

⑥ `<deployment>` is for labels on deployments (kubernetes) and deployment configs (openshift)

⑦ The subelements are also available for specifying annotations.

Labels and annotations can be specified in free form as a map. In this map the element name is the name of the label or annotation respectively, whereas the content is the value to set.

The following subelements are possible for `<labels>` and `<annotations>` :

Table 7. Label and annotation configuration

Element	Description
all	All entries specified in the `<all>` sections are applied to all resource objects created. This also implies build object like image stream and build configs which are create implicitely for an OpenShift build.
deployment	Labels and annotations applied to `Deployment` (for Kubernetes) and `DeploymentConfig` (for OpenShift) objects
pod	Labels and annotations applied pod specification as used in `ReplicationController`, `ReplicaSets`, `Deployments` and `DeploymentConfigs` objects.
replicaSet	Labels and annotations applied to `ReplicaSet` and `ReplicationController` objects.
service	Labels and annotations applied to `Service` objects.

5.1.2. Secrets

Once you've configured some docker registry credentials into `~/.m2/setting.xml`, as explained in the Authentication section, you can create Kubernetes secrets from a server declaration.

XML configuration

You can create a secret using xml configuration in the `pom.xml` file. It should contain the following fields:

key	required	description
dockerServerId	`true`	the server id which is configured in `~/.m2/setting.xml`
name	`true`	this will be used as name of the kubernetes secret resource
namespace	`false`	the secret resource will be applied to the specific namespace, if provided

This is best explained by an example.

Example for XML configuration

```
<properties>
    <docker.registry>docker.io</docker.registry>
</properties>
...
<configuration>
    <resources>
        <secrets>
            <secret>
                <dockerServerId>${docker.registry}</dockerServerId>
                <name>mydockerkey</name>
            </secret>
        </secrets>
    </resources>
</configuration>
```

Yaml fragment with annotation

You can create a secret using a yaml fragment. You can reference the docker server id with an annotation `maven.fabric8.io/dockerServerId`. The yaml fragment file should be put under the `src/main/fabric8/` folder.

Example

```
apiVersion: v1
kind: Secret
metadata:
  name: mydockerkey
  namespace: default
  annotations:
    maven.fabric8.io/dockerServerId: ${docker.registry}
type: kubernetes.io/dockercfg
```

5.1.3. Resource Validation

Resource goal also validates the generated resource descriptors using API specification of Kubernetes and OpenShift.

Table 8. Validation Configuration

Configuration	Description	Default
fabric8.skipResourceValidation	If value is set to `true` then resource validation is skipped. This may be useful if 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 printed otherwise.	`false`

Configuration	Description	Default
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`

5.1.4. Route Generation

When the `fabric8:resource` goal is run, an OpenShift Route descriptor (`route.yml`) will also be generated along the service if an OpenShift cluster is targeted. If you do not want to generate a Route descriptor, you can set the `fabric8.openshift.generateRoute` property to `false`.

Table 9. Route Generation Configuration

Configuration	Description	Default
fabric8.openshift.generateRoute	If value is set to `false` then no Route descriptor will be generated. By default it is set to `true`, which will create a `route.yml` descriptor and also add Route resource to `openshift.yml`.	`true`

If you do not want to generate a Route descriptor, you can also specify so in the plugin configuration in your POM as seen below.

Example for not generating route resource by configuring it in `pom.xml`

```
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>3.5.41</version>
    <configuration>
        <generateRoute>false</generateRoute>
    </configuration>
</plugin>
```

If you are using resource fragments, then also you can configure it in your Service resource fragment (e.g. `service.yml`). You need to add an `expose` label to the `metadata` section of your service and set it to `false`.

Example for not generating route resource by configuring it in resource fragments

```
metadata:
  annotations:
    api.service.kubernetes.io/path: /hello
  labels:
    expose: "false"
spec:
  type: LoadBalancer
```

In case both the label and the property have been set with conflicting values, precedence will be given to the property value, so if you set the label to `true` but set the property to `false` then no Route descriptor will be generated because precedence will be given to the property value.

5.1.5. Disable Automatic Deployments

Table 10. Automatic Trigger Configuration

Configuration	Description	Default
fabric8.openshift.enableAutomaticTrigger	If the value is set to `false` then automatic deployments would be disabled.	`true`

5.2. fabric8:build

This goal is for building Docker images. Images can be built in two different ways depending on the `mode` configuration (controlled by the `fabric8.mode` property). By default the mode is set to `auto`. In this case the plugin tries to detect which kind of build should be performed by contacting the API server. If this fails or if no cluster access is configured e.g. with `oc login` then the mode is set to `kubernetes` in which case a standard Docker build is performed. It can also be forced to `openshift` to perform an OpenShift build.

5.2.1. Kubernetes Build

If the mode is set to `kubernetes` then a normal Docker build is performed. The connection configuration to access the Docker daemon is described in Access Configuration.

In order to make the generated images available to the Kubernetes cluster the generated images need to be pushed to a registry with the goal **fabric8:push**. This is not necessary for single node clusters, though as there is no need to distribute images.

5.2.2. OpenShift Build

For the `openshift` mode, OpenShift specific builds will be performed. These are so called Binary Source builds ("binary builds" in short), where the data specified with the build configuration is sent directly to OpenShift as a binary archive.

There are two kind of binary builds supported by this plugin, which can be selected with the

`buildStrategy` configuration option (`fabric8.build.strategy` property)

Table 11. Build Strategies

buildStrategy	Description
s2i	The Source-to-Image (S2I) build strategy uses so called builder images for creating new application images from binary build data. The builder image to use is taken from the base image configuration specified with from in the image build configuration. See below for a list of builder images which can be used with this plugin.
docker	A Docker Build is similar to a normal Docker build except that it is done by the OpenShift cluster and not by a Docker daemon. In addition this build pushes the generated image to the OpenShift internal registry so that it is accessbile in the whole cluster.

Both build strategies update an Image Stream after the image creation.

The Build Config and Image streams can be managed by this plugin. If they do not exist, they will be automatically created by `fabric8:build`. If they do already exist, they are reused, except when the `buildRecreate` configuration option (property `fabric8.build.recreate`) is set to a value as described in Configuration. Also if the provided build strategy is different than the one defined in the existing build configuration, the Build Config is edited to reflect the new type (which in turn removes all build associated with the previous build).

This image stream created can then be directly referenced from Deployment Configuration objects created by **fabric8:resource**. By default, image streams are created with a local lookup policy, so that they can be used also by other resources such as Deployments or StatefulSets. This behavior can be turned off by setting the `fabric8.s2i.imageStreamLookupPolicyLocal` property to `false` when building the project.

In order to be able to to create these OpenShift resource objects access to an OpenShift installation is required. The access parameters are described in Access Configuration.

Regardless of which build mode is used, the images are configured in the same way.

The configuration consists of two parts: * a global section which defines the overall behaviour of this plugin * and an `<images>` section which defines how the images should be build

Many of the options below are relevant for the Kubernetes Workflow or the OpenShift Workflow with Docker builds as they influence how the Docker image is build.

For an S2I binary build, on the other hand, the most relevant section is the Assembly one because the build depends on which buider/base image is used and how it interprets the content of the uploaded `docker.tar`.

5.2.3. Configuration

The following sections describe the usual configuration, which is similar to the build configuration used in the docker-maven-plugin.

In addition a more automatic way for creating predefined build configuration can be performed with so called Generators. Generators are very flexible and can be easily created. These are described in an extra section.

Global configuration parameters specify overall behavior common for all images to build. Some of the configuration options are shared with other goals.

Table 12. Global configuration

Element	Description	Property
apiVersion	Use this variable if you are using an older version of docker not compatible with the current default use to communicate with the server.	`docker.apiVersion`
authConfig	Authentication information when pulling from or pushing to Docker registry. There is a dedicated section Authentication for how doing security.	
autoPull	Decide how to pull missing base images or images to start: * `on` : Automatic download any missing images (default) * `off` : Automatic pulling is switched off * `always` : Pull images always even when they are already exist locally * `once` : For multi-module builds images are only checked once and pulled for the whole build.	`docker.autoPull`
buildRecreate	If the effective mode is `openshift` then this option decides how the OpenShift resource objects associated with the build should be treated when they already exist: * `buildConfig` or `bc` : Only the BuildConfig is recreated * `imageStream` or `is` : Only the ImageStream is recreated * `all` : Both, BuildConfig and ImageStream are recreated * `none` : Neither BuildConfig nor ImageStream is recreated The default is `none`. If you provide the property without value then `all` is assumed, so everything gets recreated.	`fabric8.build.recreate`
buildStrategy	If the effective mode is `openshift` then this option sets the build strategy. This can be: * `s2i` for a Source-to-Image build with a binary source * `docker` for a Docker build with a binary source By default S2I is used.	`fabric8.build.strategy`
certPath	Path to SSL certificate when SSL is used for communicating with the Docker daemon. These certificates are normally stored in `~/.docker/`. With this configuration the path can be set explicitly. If not set, the fallback is first taken from the environment variable `DOCKER_CERT_PATH` and then as last resort `~/.docker/`. The keys in this are expected with it standard names `ca.pem`, `cert.pem` and `key.pem`. Please refer to the Docker documentation for more information about SSL security with Docker.	`docker.certPath`

Element	Description	Property
dockerHost	The URL of the Docker Daemon. If this configuration option is not given, then the optional `<machine>` configuration section is consulted. The scheme of the URL can be either given directly as `http` or `https` depending on whether plain HTTP communication is enabled or SSL should be used. Alternatively the scheme could be `tcp` in which case the protocol is determined via the IANA assigned port: 2375 for `http` and 2376 for `https`. Finally, Unix sockets are supported by using the scheme `unix` together with the filesystem path to the unix socket. The discovery sequence used by the docker-maven-plugin to determine the URL is: . value of **dockerHost** (`docker.host`) . the Docker host associated with the docker-machine named in `<machine>`, i.e. the `DOCKER_HOST` from `docker-machine env`. See below for more information about Docker machine support. . the value of the environment variable `DOCKER_HOST`. . `unix:///var/run/docker.sock` if it is a readable socket.	`docker.host`
image	In order to temporarily restrict the operation of plugin goals this configuration option can be used. Typically this will be set via the system property `docker.image` when Maven is called. The value can be a single image name (either its alias or full name) or it can be a comma separated list with multiple image names. Any name which doesn't refer an image in the configuration will be ignored.	`docker.image`
machine	Docker machine configuration. See Docker Machine for possible values	
mode	The build mode which can be * `kubernetes` : A Docker image will be created by calling a Docker daemon. See Kubernetes Build for details. * `openshift` : An OpenShift Build will be triggered, which can be either a *Docker binary build* or a *S2I binary build*, depending on the configuration `buildStrategy`. See OpenShift Build for details. * `auto` : The plugin tries to detect the mode by contacting the configured cluster. `auto` is the default. (*Because of technical reasons, "kubernetes" is currently the default, but will change to "auto" eventually*)	`fabric8.mode`
maxConnections	Number of parallel connections are allowed to be opened to the Docker Host. For parsing log output, a connection needs to be kept open (as well for the wait features), so don't put that number to low. Default is 100 which should be suitable for most of the cases.	`docker.maxConnections`
namespace	Namespace to use when accessing Kubernetes or OpenShift	`fabric8.namespace`
outputDirectory	Default output directory to be used by this plugin. The default value is `target/docker` and is only used for the goal `fabric8:build`.	`docker.target.dir`
portPropertyFile	Global property file into which the mapped properties should be written to. The format of this file and its purpose are also described in Port Mapping.	
profile	Profile to which contains enricher and generators configuration. See Profiles for details.	`fabric8.profile`

Element	Description	Property
pullSecret	The name to use for naming pullSecret to be created to pull the base image in case pulling from a private registry which requires authentication for Openshift. The default value for pull registry will be picked from "docker.pull.registry/docker.registry".	`fabric8.build.pullSecret`
registry	Specify globally a registry to use for pulling and pushing images. See Registry handling for details.	`docker.registry`
resourceDir	Directory where fabric8 resources are stored. This is also the directory where a custom profile is looked up. Default is `src/main/fabric8`.	`fabric8.resourceDir`
skip	With this parameter the execution of this plugin can be skipped completely.	`docker.skip`
skipBuild	If set not images will be build (which implies also *skip.tag*) with `fabric8:build`	`docker.skip.build`
skipBuildPom	If set the build step will be skipped for modules of type `pom`. If not set, then by default projects of type `pom` will be skipped if there are no image configurations contained.	`fabric8.skip.build.pom`
skipTag	If set to `true` this plugin won't add any tags to images that have been built with `fabric8:build`	`docker.skip.tag`
skipMachine	Skip using docker machine in any case	`docker.skip.machine`
sourceDirectory	Default directory that contains the assembly descriptor(s) used by the plugin. The default value is `src/main/docker`. This option is only relevant for the `fabric8:build` goal.	`docker.source.dir`
verbose	Boolean attribute for switching on verbose output like the build steps when doing a Docker build. Default is `false`	`docker.verbose`

5.2.4. Image Configuration

The configuration how images should be created a defined in a dedicated `<images>` sections. These are specified for each image within the `<images>` element of the configuration with one `<image>` element per image to use.

The `<image>` element can contain the following sub elements:

Table 13. Image Configuration

Element	Description
name	Each `<image>` configuration has a mandatory, unique docker repository *name*. This can include registry and tag parts, but also placeholder parameters. See below for a detailed explanation.
alias	Shortcut name for an image which can be used for identifying the image within this configuration. This is used when linking images together or for specifying it with the global **image** configuration element.

Element	Description
registry	Registry to use for this image. If the `name` already contains a registry this takes precedence. See Registry handling for more details.
build	Element which contains all the configuration aspects when doing a **fabric8:build**. This element can be omitted if the image is only pulled from a registry e.g. as support for integration tests like database images.

Name placeholders

When specifying the name you can use several placeholders which are replaced during runtime by this plugin. In addition you can use regular Maven properties which are resolved by Maven itself.

Table 14. Placeholders

Placeholder	Description
%g	The last part of the Maven group name, sanitized so that it can be used as username on GitHub. Only the part after the last dot is used. E.g. for a group id `io.fabric8` this placeholder would insert `fabric8`
%a	A sanitized version of the artefact id so that it can be used as part of an Docker image name. I.e. it is converted to all lower case (as required by Docker)
%v	The project version. Synonym to `${project.version}`
%l	If the project version ends with `-SNAPSHOT` then this placeholder is `latest`, otherwise its the full version (same as `%v`)
%t	If the project version ends with `-SNAPSHOT` this placeholder resolves to `snapshot-<timestamp>` where timestamp has the date format `yyMMdd-HHmmss-SSSS` (eg `snapshot-`). This feature is especially useful during development in oder to avoid conflicts when images are to be updated which are still in use. You need to take care yourself of cleaning up old images afterwards, though.

The `<build>` section is mandatory and is explained in below.

Example for <image>

```
<configuration>
  ....
  <images>
    <image> ①
      <name>%g/docker-demo:0.1</name> ②
      <alias>service</alias> ③
      <build>....</build> ④
    </image>
    <image>
      ....
    </image>
  </images>
</configuration>
```

① One or more `<image>` definitions

② The Docker image name used when creating the image.

③ An alias which can be used in other parts of the plugin to reference to this image. This alias must be unique.

④ A `<build>` section as described in Build Configuration

5.2.5. Build Configuration

There are two different modes how Images can be built:

Inline plugin configuration

With an inline plugin configuration all information required to build the image is contained in the plugin configuration. By default its the standard XML based configuration for the plugin but can be switched to a property based configuration syntax as described in the section External configuration. The XML configuration syntax is recommended because of its more structured and typed nature.

When using this mode, the Dockerfile is created on the fly with all instructions extracted from the configuration given.

External Dockerfile or Docker archive

Alternatively an external Dockerfile template or Docker archive can be used. This mode is switched on by using one of these three configuration options within

- **dockerFileDir** specifies a directory containing a Dockerfile that will be used to create the image. The name of the Dockerfile is `Dockerfile` by default but can be also set with the option `dockerFile` (see below).
- **dockerFile** specifies a specific Dockerfile path. The Docker build context directory is set to `dockerFileDir` if given. If not the directory by default is the directory in which the Dockerfile is stored.
- **dockerArchive** specifies a previously saved image archive to load directly. Such a tar archive can be created with `docker save`. If a `dockerArchive` is provided, no `dockerFile` or `dockerFileDir` must be given.

All paths can be either absolute or relative paths (except when both `dockerFileDir` and `dockerFile` are provided in which case `dockerFile` must not be absolute). A relative path is looked up in `${project.basedir}/src/main/docker` by default. You can make it easily an absolute path by using `${project.basedir}` in your configuration.

Adding assemblies in Dockerfile mode

Any additional files located in the `dockerFileDir` directory will also be added to the build context as well. You can also use an assembly if specified in an assembly configuration. However you need to add the files on your own in the Dockerfile with an `ADD` or `COPY` command. The files of the assembly are stored in a build context relative directory `maven/` but can be changed by changing the assembly name with the option `<name>` in the assembly configuration.

E.g. the files can be added with

Example

```
COPY maven/ /my/target/directory
```

so that the assembly files will end up in `/my/target/directory` within the container.

If this directory contains a `.maven-dockerignore` (or alternatively, a `.maven-dockerexclude` file), then it is used for excluding files for the build. Each line in this file is treated as an FileSet exclude pattern as used by the maven-assembly-plugin. It is similar to `.dockerignore` when using Docker but has a slightly different syntax (hence the different name).

If this directory contains a `.maven-dockerinclude` file, then it is used for including only those files for the build. Each line in this file is also treated as an FileSet exclude pattern as used by the maven-assembly-plugin.

Except for the assembly configuration all other configuration options are ignored for now.

Filtering

fabric8-maven-plugin filters given Dockerfile with Maven properties, much like the `maven-resource-plugin` does. Filtering is enabled by default and can be switched off with a build config `<filter>false</filter>`. Properties which we want to replace are specified with the `${..}` syntax. Only properties which are set in the Maven build are replaced, all other remain untouched.

This partial replacement means that you can easily mix it with Docker build arguments and environment variable reference, but you need to be careful. If you want to be more explicit about the property delimiter to clearly separate Docker properties and Maven properties you can redefine the delimiter. In general, the `filter` option can be specified the same way as delimiters in the resource plugin. In particular, if this configuration contains a * then the parts left, and right of the asterisks are used as delimiters.

For example, the default `<filter>${*}</filter>` parse Maven properties in the format that we know. If you specify a single character for `<filter>` then this delimiter is taken for both, the start and the end. E.g a `<filter>@</filter>` triggers on parameters in the format `@···@`, much like in the `maven-invoker-plugin`. Use something like this if you want to clearly separate from Docker builds args. Property replacement works for Dockerfile only. For replacing other data in associated files targeted for the Docker image, please use the maven-resource-plugin to copy them over before.

Example

The following example uses a Dockerfile in the directory `src/main/docker/demo` and replaces all properties in the format `@property@` within the Dockerfile.

```
<plugin>
 <configuration>
   <images>
     <image>
       <name>user/demo</name>
       <build>
         <dockerFileDir>demo</dockerFileDir>
         <filter>@</filter>
       </build>
     </image>
   </images>
 </configuration>
 ...
</plugin>
```

All build relevant configuration is contained in the `<build>` section of an image configuration. The following configuration options are supported:

Table 15. Build configuration (<image>)

Element	Description
assembly	specifies the assembly configuration as described in Build Assembly
buildArgs	Map specifying the value of Docker build args which should be used when building the image with an external Dockerfile which uses build arguments. The key-value syntax is the same as when defining Maven properties (or `labels` or `env`). This argument is ignored when no external Dockerfile is used. Build args can also be specified as properties as described in Build Args
buildOptions	Map specifying the build options to provide to the docker daemon when building the image. These options map to the ones listed as query parameters in the Docker Remote API and are restricted to simple options (e.g.: memory, shmsize). If you use the respective configuration options for build options natively supported by the build configuration (i.e. `nocache`, `cleanup` for `forcerm=1` and `buildArgs`) then these will override any corresponding options given here. The key-value syntax is the same as when defining environment variables or labels as described in Setting Environment Variables and Labels.
cleanup	Cleanup dangling (untagged) images after each build (including any containers created from them). Default is `try` which tries to remove the old image, but doesn't fail the build if this is not possible because e.g. the image is still used by a running container. Use `remove` if you want to fail the build and `none` if no cleanup is requested.
cmd	A command to execute by default (i.e. if no command is provided when a container for this image is started). See Startup Arguments for details.
compression	The compression mode how the build archive is transmitted to the docker daemon (`fabric8:build`) and how docker build archives are attached to this build as sources (`fabric8:source`). The value can be `none` (default), `gzip` or `bzip2`.

Element	Description
dockerFile	Path to a `Dockerfile` which also triggers *Dockerfile mode*. See External Dockerfile for details.
dockerFileDir	Path to a directory holding a `Dockerfile` and switch on *Dockerfile mode*. See External Dockerfile for details.
dockerArchive	Path to a saved image archive which is then imported. See Docker archive for details.
entryPoint	An entrypoint allows you to configure a container that will run as an executable. See Startup Arguments for details.
env	The environments as described in Setting Environment Variables and Labels.
filter	Enable and set the delimiters for property replacements. By default properties in the format `${..}` are replaced with Maven properties. You can switch off property replacement by setting this property to `false`. When using a single char like `@` then this is used as a delimiter (e.g `@…@`). See Filtering for more details.
from	The base image which should be used for this image. If not given this default to `busybox:latest` and is suitable for a pure data image. In case of an S2I Binary build this parameter specifies the S2I Builder Image to use, which by default is `fabric8/s2i-java:latest`. See also from-ext how to add additional properties for the base image.
fromExt	Extended definition for a base image. This field holds a map of defined in `<key>value</key>` format. The known keys are: * `<name>` : Name of the base image * `<kind>` : Kind of the reference to the builder image when in S2I build mode. By default its `ImageStreamTag` but can be also `ImageStream`. An alternative would be `DockerImage` * `<namespace>` : Namespace where this builder image lives. A provided `<from>` takes precedence over the name given here. This tag is useful for extensions of this plugin like the fabric8-maven-plugin which can evaluate the additional information given here.
healthCheck	Definition of a health check as described in Healthcheck
labels	Labels as described in Setting Environment Variables and Labels.
maintainer	The author (`MAINTAINER`) field for the generated image
nocache	Don't use Docker's build cache. This can be overwritten by setting a system property `docker.nocache` when running Maven.
optimise	if set to true then it will compress all the `runCmds` into a single `RUN` directive so that only one image layer is created.
ports	The exposed ports which is a list of `<port>` elements, one for each port to expose. Whitespace is trimmed from each element and empty elements are ignored. The format can be either pure numerical ("8080") or with the protocol attached ("8080/tcp").

Element	Description
runCmds	Commands to be run during the build process. It contains **run** elements which are passed to the shell. Whitespace is trimmed from each element and empty elements are ignored. The run commands are inserted right after the assembly and after **workdir** in to the Dockerfile. This tag is not to be confused with the `<run>` section for this image which specifies the runtime behaviour when starting containers.
skip	if set to true disables building of the image. This config option is best used together with a maven property
tags	List of additional `tag` elements with which an image is to be tagged after the build. Whitespace is trimmed from each element and empty elements are ignored.
user	User to which the Dockerfile should switch to the end (corresponds to the `USER` Dockerfile directive).
volumes	List of `volume` elements to create a container volume. Whitespace is trimmed from each element and empty elements are ignored.
workdir	Directory to change to when starting the container.

From this configuration this Plugin creates an in-memory Dockerfile, copies over the assembled files and calls the Docker daemon via its remote API.

Example

```
<build>
  <from>java:8u40</from>
  <maintainer>john.doe@example.com</maintainer>
  <tags>
    <tag>latest</tag>
    <tag>${project.version}</tag>
  </tags>
  <ports>
    <port>8080</port>
  </ports>
  <volumes>
    <volume>/path/to/expose</volume>
  </volumes>
  <buildOptions>
    <shmsize>2147483648</shmsize>
  </buildOptions>

  <entryPoint>
    <!-- exec form for ENTRYPOINT -->
    <exec>
      <arg>java</arg>
      <arg>-jar</arg>
      <arg>/opt/demo/server.jar</arg>
    </exec>
  </entryPoint>

  <assembly>
    <mode>dir</mode>
    <targetDir>/opt/demo</targetDir>
    <descriptor>assembly.xml</descriptor>
  </assembly>
</build>
```

In order to see the individual build steps you can switch on `verbose` mode either by setting the property `docker.verbose` or by using `<verbose>true</verbose>` in the Global configuration

5.2.6. Assembly

The `<assembly>` element within `<build>` is has an XML struture and defines how build artifacts and other files can enter the Docker image.

Table 16. Assembly Configuration (<image> : <build>)

Element	Description
name	Assembly name, which is `maven` by default. This name is used for the archived and directories created during the build and holding the assembly files. If an external Dockerfile is used than this name is also the relative directory which contains the assembly files.

Element	Description
targetDir	Directory under which the files and artifacts contained in the assembly will be copied within the container. The default value for this is `/<assembly name>`, so `/maven` if **name** is not set to a different value. This option has no meaning when an external Dockerfile is used.
inline	Inlined assembly descriptor as described in Assembly Descriptor below.
descriptor	Path to an assembly descriptor file, whose format is described Assembly Descriptor below.
descriptorRef	Alias to a predefined assembly descriptor. The available aliases are also described in Assembly Descriptor below.
dockerFileDir	Directory containing an external Dockerfile. *This option is deprecated, please use <dockerfiledir> directly in the <build> section.*
exportTargetDir	Specification whether the `targetDir` should be exported as a volume. This value is `true` by default except in the case the `targetDir` is set to the container root (`/`). It is also `false` by default when a base image is used with `from` since exporting makes no sense in this case and will waste disk space unnecessarily.
ignorePermissions	Specification if existing file permissions should be ignored when creating the assembly archive with a mode `dir`. This value is `false` by default. *This property is deprecated, use a* `permissionMode` *of* `ignore` *instead.*
mode	Mode how the how the assembled files should be collected: * `dir` : Files are simply copied (default), * `tar` : Transfer via tar archive * `tgz` : Transfer via compressed tar archive * `zip` : Transfer via ZIP archive The archive formats have the advantage that file permission can be preserved better (since the copying is independent from the underlying files systems), but might triggers internal bugs from the Maven assembler (as it has been reported in #171)
permissions	Permission of the files to add: * `ignore` to use the permission as found on files regardless on any assembly configuration * `keep` to respect the assembly provided permissions, `exec` for setting the executable bit on all files (required for Windows when using an assembly mode `dir`) * `auto` to let the plugin select `exec` on Windows and `keep` on others. `keep` is the default value.
tarLongFileMode	Sets the TarArchiver behaviour on file paths with more than 100 characters length. Valid values are: "warn"(default), "fail", "truncate", "gnu", "posix", "posix_warn" or "omit"

Element	Description
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. If a third part is given, then the build changes to user `root` before changing the ownerships, changes the ownerships and then change to user `run-user` which is then used for the final command to execute. This feature might be needed, if the base image already changed the user (e.g. to 'jboss') so that a `chown` from root to this user would fail. For example, the image `jboss/wildfly` use a "jboss" user under which all commands are executed. Adding files in Docker always happens under the UID root. These files can only be changed to "jboss" is the `chown` command is executed as root. For the following commands to be run again as "jboss" (like the final `standalone.sh`), the plugin switches back to user `jboss` (this is this "run-user") after changing the file ownership. For this example a specification of `jboss:jboss:jboss` would be required.

In the event you do not need to include any artifacts with the image, you may safely omit this element from the configuration.

Assembly Descriptor

With using the `inline`, `descriptor` or `descriptorRef` option it is possible to bring local files, artifacts and dependencies into the running Docker container. A `descriptor` points to a file describing the data to put into an image to build. It has the same format as for creating assemblies with the maven-assembly-plugin with following exceptions:

- `<formats>` are ignored, the assembly will allways use a directory when preparing the data container (i.e. the format is fixed to `dir`)
- The `<id>` is ignored since only a single assembly descriptor is used (no need to distinguish multiple descriptors)

Also you can inline the assembly description with a `inline` description directly into the pom file. Adding the proper namespace even allows for IDE autocompletion. As an example, refer to the profile `inline` in the `data-jolokia-demo` 's pom.xml.

Alternatively `descriptorRef` can be used with the name of a predefined assembly descriptor. The following symbolic names can be used for `descriptorRef`:

Table 17. Predefined Assembly Descriptors

Assembly Reference	Description
artifact-with-dependencies	Attaches project's artifact and all its dependencies. Also, when a `classpath` file exists in the target directory, this will be added to.
artifact	Attaches only the project's artifact but no dependencies.
project	Attaches the whole Maven project but with out the `target/` directory.
rootWar	Copies the artifact as `ROOT.war` to the exposed directory. I.e. Tomcat will then deploy the war under the root context.

Example

```
<images>
  <image>
    <build>
      <assembly>
         <descriptorRef>artifact-with-dependencies</descriptorRef>
         .....
```

will add the created artifact with the name `${project.build.finalName}.${artifact.extension}` and all jar dependencies in the the `targetDir` (which is `/maven` by default).

All declared files end up in the configured `targetDir` (or `/maven` by default) in the created image.

Maven peculiarities when including the artifact

If the assembly references the artifact to build with this pom, it is required that the `package` phase is included in the run. Otherwise the artifact file can't be found by `docker:build`. This is an old outstanding issue of the assembly plugin which probably can't be fixed because of the way how Maven works. We tried hard to workaround this issue and in 90% of all cases you won't experience any problem. However, when the following warning happens which might lead to the given error:

```
[WARNING] Cannot include project artifact: io.fabric8:helloworld:jar:0.20.0; it
doesn't have an associated file or directory.
[WARNING] The following patterns were never triggered in this artifact inclusion
filter:
o  'io.fabric8:helloworld'

[ERROR] DOCKER> Failed to create assembly for docker image  (with mode 'dir'): Error
creating assembly archive docker: You must set at least one file.
```

then you have two options to fix this:

- Call `mvn package fabric8:build` to explicitly run "package" and "docker:build" in a chain.
- Bind `build` to an to an execution phase in the plugin's definition. By default `fabric8:build` will bind to the `install` phase is set in an execution. Then you can use a plain `mvn install` for building the artifact and creating the image.

```
<executions>
  <execution>
    <id>docker-build</id>
    <goals>
       <goal>build</goal>
    </goals>
  </execution>
<executions>
```

Example

In the following example a dependency from the pom.xml is included and mapped to the name `jolokia.war`. With this configuration you will end up with an image, based on `busybox` which has a directory `/maven` containing a single file `jolokia.war`. This volume is also exported automatically.

```
<assembly>
  <inline>
    <dependencySets>
      <dependencySet>
        <includes>
          <include>org.jolokia:jolokia-war</include>
        </includes>
        <outputDirectory>.</outputDirectory>
        <outputFileNameMapping>jolokia.war</outputFileNameMapping>
      </dependencySet>
    </dependencySets>
  </inline>
</assembly>
```

Another container can now connect to the volume an 'mount' the `/maven` directory. A container from `consol/tomcat-7.0` will look into `/maven` and copy over everything to `/opt/tomcat/webapps` before starting Tomcat.

If you are using the `artifact` or `artifact-with-dependencies` descriptor, it is possible to change the name of the final build artifact with the following:

Example

```
<build>
  <finalName>your-desired-final-name</finalName>
  ...
</build>
```

Please note, based upon the following documentation listed here, there is no guarantee the plugin creating your artifact will honor it in which case you will need to use a custom descriptor like above to achieve the desired naming.

Currently the `jar` and `war` plugins properly honor the usage of `finalName`.

5.2.7. Environment and Labels

When creating a container one or more environment variables can be set via configuration with the `env` parameter

Example

```
<env>
  <JAVA_HOME>/opt/jdk8</JAVA_HOME>
  <CATALINA_OPTS>-Djava.security.egd=file:/dev/./urandom</CATALINA_OPTS>
</env>
```

If you put this configuration into profiles you can easily create various test variants with a single image (e.g. by switching the JDK or whatever).

It is also possible to set the environment variables from the outside of the plugin's configuration with the parameter `envPropertyFile`. If given, this property file is used to set the environment variables where the keys and values specify the environment variable. Environment variables specified in this file override any environment variables specified in the configuration.

Labels can be set inline the same way as environment variables:

Example

```
<labels>
   <com.example.label-with-value>foo</com.example.label-with-value>
   <version>${project.version}</version>
   <artifactId>${project.artifactId}</artifactId>
</labels>
```

5.2.8. Startup Arguments

Using `entryPoint` and `cmd` it is possible to specify the entry point or cmd for a container.

The difference is, that an `entrypoint` is the command that always be executed, with the `cmd` as argument. If no `entryPoint` is provided, it defaults to `/bin/sh -c` so any `cmd` given is executed with a shell. The arguments given to `docker run` are always given as arguments to the `entrypoint`, overriding any given `cmd` option. On the other hand if no extra arguments are given to `docker run` the default `cmd` is used as argument to `entrypoint`.

> See this stackoverflow question for a detailed explanation.

An entry point or command can be specified in two alternative formats:

Table 18. Entrypoint and Command Configuration

Mode	Description
shell	Shell form in which the whole line is given to `shell -c` for interpretation.
exec	List of arguments (with inner `<args>`) arguments which will be given to the `exec` call directly without any shell interpretation.

Either shell or params should be specified.

Example

```
<entryPoint>
   <!-- shell form  -->
   <shell>java -jar $HOME/server.jar</shell>
</entryPoint>
```

or

Example

```
<entryPoint>
   <!-- exec form  -->
   <exec>
     <arg>java</arg>
     <arg>-jar</arg>
     <arg>/opt/demo/server.jar</arg>
   </exec>
</entryPoint>
```

This can be formulated also more dense with:

Example

```
<!-- shell form  -->
<entryPoint>java -jar $HOME/server.jar</entryPoint>
```

or

Example

```
<entryPoint>
  <!-- exec form  -->
  <arg>java</arg>
  <arg>-jar</arg>
  <arg>/opt/demo/server.jar</arg>
</entryPoint>
```

INFO

Startup arguments are not used in S2I builds

5.2.9. Build Args

As described in section Configuration for external Dockerfiles Docker build arg can be used. In addition to the configuration within the plugin configuration you can also use properties to specify them:

- Set a system property when running Maven, eg.:

`-Ddocker.buildArg.http_proxy=http://proxy:8001`. This is especially useful when using predefined Docker arguments for setting proxies transparently.

- Set a project property within the `pom.xml`, eg.:

Example

```
<docker.buildArg.myBuildArg>myValue</docker.buildArg.myBuildArg>
```

Please note that the system property setting will always override the project property. Also note that for all properties which are not Docker predefined properties, the external Dockerfile must contain an `ARGS` instruction.

5.3. fabric8:push

Section needs review and rearrangments

This goal uploads images to the registry which have a `<build>` configuration section. The images to push can be restricted with with the global option `filter` (see Global Configuration for details). The registry to push is by default `docker.io` but can be specified as part of the images's `name` name the Docker way. E.g. `docker.test.org:5000/data:1.5` will push the image `data` with tag `1.5` to the registry `docker.test.org` at port `5000`. Security information (i.e. user and password) can be specified in multiple ways as described in section Authentication.

By default a progress meter is printed out on the console, which is omitted when using Maven in batch mode (option `-B`). A very simplified progress meter is provided when using no color output (i.e. with `-Ddocker.useColor=false`).

Table 19. Push options

Element	Description	Property
skipPush	If set to `true` the plugin won't push any images that have been built.	`docker.skip.push`
pushRegistry	The registry to use when pushing the image. See Registry Handling for more details.	`docker.push.registry`
retries	How often should a push be retried before giving up. This useful for flaky registries which tend to return 500 error codes from time to time. The default is 0 which means no retry at all.	`docker.push.retries`

5.4. fabric8:apply

This goals applies the resources created with **fabric8:resource** to a connected Kubernetes or OpenShift cluster. It's similar to **fabric8:deploy** but does not the full deployment cycle of creating the resource, creating the application image and the sending the resource descriptors to the clusters. This goal can be easily bound to `<executions>` within the plugin's configuration and binds by default to the `install` lifecycle phase.

```
mvn fabric8:apply
```

5.5. fabric8:resource-apply

This goal will generate the kubernetes resources via the **fabric8:resource** goal and apply them into the current kubernetes cluster.

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

Its usually simpler to just use the **fabric8:deploy** goal which performs a build, creates the docker image and runs `fabric8:resource-apply`:

```
mvn fabric8:deploy
```

However if you have built your code and docker image but find some issue with the generated manifests; you can update the configuration of the **fabric8:resource** goal in your `pom.xml` or modify the YAML files in `src/main/fabric8` and then run:

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

Which will skip running unit tests and generating the docker build via **fabric8:build** but will only regenerate the manifests and apply them. This can help speed up the round trip time when fixing up resource generation issues.

Note to use this goal you must have the **fabric8:resource** goal bound to your executions in your pom.xml. e.g. like this:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.41</version>

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

5.6. fabric8:helm

This goal is deprecated and might vanish in version 4.0 of this plugin. It's not yet decided whether we fix this goal for 4.0 or drop it. Please raise your voice if you want to keep it. Even better, if you want to support this goal, we are always looking for contributions ;-)

This goal is for creating Helm charts for your Maven project so that you can install, update or delete your app in Kubernetes using Helm.

For creating a Helm chart you simply call `fabric8:helm` goal on the command line:

```
mvn fabric8:resource fabric8:helm
```

The `fabric8:resource` goal is required to create the resource descriptors which are included in the Helm chart. If you have already built the resource then you can ommit this goal.

The configuration happens in a `<helm>` section within the plugin's configuration:

Example Helm configuration

```
<plugin>
  <configuration>
    <helm>
      <chart>Jenkins</chart>
      <keywords>ci,cd,server</keywords>
    </helm>
    ...
  </configuration>
</plugin>
```

This configuration section know the following subelements in order to configure your Helm chart.

Table 20. Helm configuration

Element	Description	Property
chart	The Chart name, which is `${project.artifactId}` if not given.	`fabric8.helm.chart`
type	For which platform to generate the chart. By default this is `kubernetes`, but can be also `openshift` for using OpenShift specific resources in the chart. *Please note that there is no OpenShift support yet for charts, so this is experimental*. You can also add both values as a comma separated list.	`fabric8.helm.type`
sourceDir	Where to find the resource descriptors generated with `fabric8:resource`. By default this is `${basedir}/target/classes/META-INF/fabric8`, which is also the output directory used by `fabric8:resource`.	`fabric8.helm.sourceDir`

Element	Description	Property
outputDir	Where to create the the Helm chart, which is `${basedir}/target/fabric8/helm` by default for Kubernetes (and `${basedir}/target/fabric8/helmshift` for OpenShift).	`fabric8.helm.outputDir`
keywords	Comma separated list of keywords to add to the chart	
engine	The template engine to use	
chartExtension	The Helm chart file extension, default value is `tar.gz`	`fabric8.helm.chartExtension`

In a next step you can install this via the helm command line tool as follows:

```
helm install target/fabric8/helm/kubernetes
```

To add the `helm` goal to your project so that it is automatically executed just add the `helm` goal to the `executions` section of the `fabric8-maven-plugin` section of your `pom.xml`.

Add helm goal

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

  <!-- ... -->

  <executions>
    <execution>
      <goals>
        <goal>resource</goal>
        <goal>helm</goal>
        <goal>build</goal>
        <goal>deploy</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

In addition this goal will also create a tar-archive below `${basedir}/target` which contains the chart with its template. This tar is added as an aritfact with classifier `helm` to the build (`helmshift` for the OpenShift mode).

Chapter 6. Development Goals

6.1. fabric8:deploy

This is the main goal for building your docker image, generating the kubernetes resources and deploying them into the cluster (insofar your pom.xml is set up correct; keep reading :)).

```
mvn fabric8:deploy
```

This goal is designed to run **fabric8:build** and **fabric8:resource** before the deploy ***iff** you have the goals bound in your pom.xml:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.41</version>

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

Effectively this builds your project then invokes these goals:

- **fabric8:build**
- **fabric8:resource-apply**

By default the behaviour of resource goal is it generates `route.yml` for a service if you have not done any configuration changes. Sometimes there may be case when you want to generate route.yml but do not want to create route resource on OpenShift Cluster. This can be achieved by the following configuration.

Example for not generating route resource on your cluster

```
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>3.5.41</version>
    <configuration>
        <enricher>
            <excludes>
                <exclude>f8-expose</exclude>
            </excludes>
        </enricher>
    </configuration>
</plugin>
```

6.2. fabric8:undeploy

This goal is for deleting the kubernetes resources that you deployed via the [fabric8:run] or **fabric8:deploy** goals

It iterates through all the resources generated by the **fabric8:resource** goal and deletes them from your current kubernetes cluster.

```
mvn fabric8:undeploy
```

6.3. fabric8:log

This goal tails the log of the app that you deployed via the **fabric8:deploy** goal

```
mvn fabric8:log
```

You can then terminate the output by hitting `Ctrl+C`

If you wish to get the log of the app and then terminate immediately then try:

```
mvn fabric8:log -Dfabric8.log.follow=false
```

This lets you pipe the output into grep or some other tool

```
mvn fabric8:log -Dfabric8.log.follow=false | grep Exception
```

If your app is running in multiple pods you can configure the pod name to log via the `fabric8.log.pod` property, otherwise it defaults to the latest pod:

```
mvn fabric8:log -Dfabric8.log.pod=foo
```

If your pod has multiple containers you can configure the container name to log via the `fabric8.log.container` property, otherwise it defaults to the first container:

```
mvn fabric8:log -Dfabric8.log.container=foo
```

6.4. fabric8:debug

This goal enables debugging in your Java app and then port forwards from localhost to the latest running pod of your app so that you can easily debug your app from your Java IDE.

```
mvn fabric8:debug
```

Then follow the on screen instructions.

The default debug port is `5005`. If you wish to change the local port to use for debugging then pass in the `fabric8.debug.port` parameter:

```
mvn fabric8:debug -Dfabric8.debug.port=8000
```

Then in your IDE you start a Remote debug execution using this remote port using localhost and you should be able to set breakpoints and step through your code.

This lets you debug your apps while they are running inside a Kubernetes cluster - for example if you wish to debug a REST endpoint while another pod is invoking it.

Debug is enabled via the `JAVA_ENABLE_DEBUG` environment variable being set to `true`. This environment variable is used for all the standard Java docker images used by Spring Boot, flat classpath and executable JAR projects and Wildfly Swarm. If you use your own custom docker base image you may wish to also respect this environment variable too to enable debugging.

6.4.1. Speeding up debugging

By default the `fabric8:debug` goal has to edit your Deployment to enable debugging then wait for a pod to start. It might be in development you frequently want to debug things and want to speed things up a bit.

If so you can enable debug mode for each build via the `fabric8.debug.enabled` property.

e.g. you can pass this property on the command line:

```
mvn fabric8:deploy -Dfabric8.debug.enabled=true
```

Or you can add something like this to your `~/.m2/settings.xml` file so that you enable debug mode for all maven builds on your laptop by using a profile :

```
<?xml version="1.0"?>
<settings>
  <profiles>
    <profile>
      <id>enable-debug</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
        <fabric8.debug.enabled>true</fabric8.debug.enabled>
      </properties>
    </profile>
```

Then whenever you type the `fabric8:debug` goal there is no need for the maven goal to edit the `Deployment` and wait for a pod to restart; we can immediately start debugging when you type:

```
mvn fabric8:debug
```

6.4.2. Debugging with suspension

The `fabric8:debug` goal allows to attach a remote debugger to a running container, but the application is free to execute when the debugger is not attached. In some cases, you may want to have complete control on the execution, e.g. to investigate the application behavior at startup. This can be done using the `fabric8.debug.suspend` flag:

```
mvn fabric8:debug -Dfabric8.debug.suspend
```

The suspend flag will set the `JAVA_DEBUG_SUSPEND` environment variable to `true` and `JAVA_DEBUG_SESSION` to a random number in your deployment. When the `JAVA_DEBUG_SUSPEND` environment variable is set, standard docker images will use `suspend=y` in the JVM startup options for debugging.

The `JAVA_DEBUG_SESSION` environment variable is always set to a random number (each time you run the debug goal with the suspend flag) in order to tell Kubernetes to restart the pod. The remote application will start only after a remote debugger is attached. You can use the remote debugging feature of your IDE to connect (on `localhost`, port `5005` by default).

The `fabric8.debug.suspend` flag will disable readiness probes in the Kubernetes deployment in order to start port-forwarding during the early phases of application startup

6.5. fabric8:watch

This goal is used to monitor the project workspace for changes and automatically trigger a redeploy of the application running on Kubernetes.

Before entering the watch mode, this goal must generate the docker image and the Kubernetes resources (optionally including some development libraries/configuration), and deploy the app on Kubernetes. Lifecycle bindings should be configured as follows to allow the generation of such resources.

Lifecycle bindings for fabric8:watch

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

  <!-- ... -->

  <executions>
    <execution>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

For any application having `resource` and `build` goals bound to the lifecycle, the following command can be used to run the watch task.

```
mvn fabric8:watch
```

This plugin supports different watcher providers, enabled automatically if the project satisfies certain conditions.

Watcher providers can also be configured manually. The Generator example is a good blueprint, simply replace `<generator>` with `<watcher>`. The configuration is structurally identical.

6.5.1. Spring Boot

This watcher is enabled by default for all Spring Boot projects. It performs the following actions:

- deploys your application with Spring Boot DevTools enabled
- tails the log of the latest running pod for your application
- watches the local development build of your Spring Boot based application and then triggers a reload of the application when there are changes

You can try it on any spring boot application via:

```
mvn fabric8:watch
```

Once the goal starts up the spring boot RemoteSpringApplication it will watch for local development changes.

e.g. if you edit the java code of your app and then build it via something like this:

```
mvn package
```

You should see your app reload on the fly in the shell running the **fabric8:watch** goal!

There is also support for LiveReload as well.

6.5.2. Docker Image

This is a generic watcher that can be used in Kubernetes mode only. Once activated, it listens for changes in the project workspace in order to trigger a redeploy of the application.

The watcher can be activated e.g. by running this command in another shell:

```
mvn package
```

The watcher will detect that the binary artifact has changed and will first rebuild the docker image, then start a redeploy of the Kubernetes pod.

It uses the watch feature of the docker-maven-plugin under the hood.

Chapter 7. Generators

The usual way to define Docker images is with the plugin configuration as explained in **fabric8:build**. This can either be done completely within the `pom.xml` or by referring to an external Dockerfile. Since fabric8-maven-plugin includes docker-maven-plugin the way by which images are built is identical.

However, this plugin provides an additional route for defining image configurations. This is done by so called *Generators*. A generator is a Java component providing an auto-detection mechanism for certain build types like a Spring Boot build or a plain Java build. As soon as a *Generator* detects that it is applicable it will be called with the list of images configured in the `pom.xml`. Typically a generator only creates dynamically a new image configuration if this list is empty. But a generator is free to also add new images to an existing list or even change the current image list.

You can easily create your own generator as explained in Generator API. This section will focus on existing generators and how you can configure them.

The included *Generators* are enabled by default, but you can easily disable them or only select a certain set of generators. Each generator has a *name*, which is unique for a generator.

The generator configuration is embedded in a `<generator>` configuration section:

Example for a generator configuration

```
<plugin>
  ....
  <configuration>
    ....
    <generator> ①
      <includes> ②
        <include>spring-boot</include>
      </includes>
      <config> ③
        <spring-boot> ④
          <alias>ping</alias>
        </spring-boot>
      </config>
    </generator>
  </configuration>
</plugin>
```

① Start of generators' configuration.

② Generators can be included and excluded. Includes have precedence, and the generators are called in the given order.

③ Configuration for individual generators.

④ The config is a map of supported config values. Each section is embedded in a tag named after the generator.

The following sub-elements are supported:

Table 21. Generator configuration

Element	Description
`<includes>`	Contains one ore more `<include>` elements with generator names which should be included. If given only this list of generators are included in this given order. The order is important because by default only the first matching generator kicks in. The generators from every active profile are included, too. However the generators listed here are moved to the front of the list, so that they are called first. Use the profile `raw` if you want to explicitly set the complete list of generators.
`<excludes>`	Holds one or more `<exclude>` elements with generator names to exclude. If set then all detected generators are used except the ones mentioned in this section.
`<config>`	Configuration for all generators. Each generator support a specific set of configuration values as described in the documentation. The subelements of this section are generator names to configure. E.g. for generator `spring-boot`, the sub-element is called `<spring-boot>`. This element then holds the specific generator configuration like `<name>` for specifying the final image name. See above for an example. Configuration coming from profiles are merged into this config, but not overriding the configuration specified here.

Beside specifying generator configuration in the plugin's configuration it can be set directly with properties, too:

Example generator property config

```
mvn -Dfabric8.generator.spring-boot.alias="myapp"
```

The general scheme is a prefix `fabric8.generator.` followed by the unique generator name and then the generator specific key.

In addition to the provided default *Generators* described in the next section Default Generators, custom generators can be easily added. There are two ways to include generators:

Plugin dependency

You can declare the generator holding jars as dependency to this plugin as shown in this example

```
<plugin>
  <artifactId>fabric8-maven-plugin</artifactId>
  ....
  <dependencies>
    <dependency>
      <groupId>io.acme</groupId>
      <artifactId>mygenerator</artifactId>
      <version>1.0</version>
    <dependency>
  </dependencies>
</plugin>
```

Compile time dependency

Alternatively and if your application code comes with a custom generator you can set the global configuration option `useProjectClasspath` (property: `fabric8.useProjectClasspath`) to true. In this case also the project artifact and its dependencies are looked up for *Generators*. See Generator API for details how to write your own generators.

7.1. Default Generators

All default generators examine the build information for certain aspects and generate a Docker build configuration on the fly. They can be configured to a certain degree, where the configuration is generator specific.

Table 22. Default Generators

Generator	Name	Description
Java Applications	`java-exec`	Generic generator for flat classpath and fat-jar Java applications
Spring Boot	`spring-boot`	Spring Boot specific generator
Wildfly Swarm	`wildfly-swarm`	Generator for Wildfly Swarm apps
Thorntail v2	`thorntail-v2`	Generator for Thorntail v2 apps
Vert.x	`vertx`	Generator for Vert.x applications
Karaf	`karaf`	Generator for Karaf based appps
Web applications	`webapps`	Generator for WAR based applications supporting Tomcat, Jetty and Wildfly base images

There are some configuration options which are shared by all generators:

Table 23. Common generator options

Element	Description	Property
add	When this set to `true`, then the generator *adds* to an existing image configuration. By default this is disabled, so that a generator only kicks in when there are no other image configurations in the build, which are either configured directly for a `fabric8:build` or already added by a generator which has been run previously.	
alias	An alias name for referencing this image in various other parts of the configuration. This is also used in the log output. The default alias name is the name of the generator.	`fabric8.generator.alias`
from	This is the base image from where to start when creating the images. By default the generators make an opinionated decision for the base image which are described in the respective generator section.	`fabric8.generator.from`

Element	Description	Property
fromMode	Whe using OpenShift S2I builds the base image can be either a plain docker image (mode: `docker`) or a reference to an ImageStreamTag (mode: `istag`). In the case of an ImageStreamTag, `from` has to be specified in the form `namespace/image-stream:tag`. The mode takes only effect when running in OpenShift mode.	`fabric8.generator.fromMode`
name	The Docker image name used when doing Docker builds. For OpenShift S2I builds its the name of the image stream. This can be a pattern as descibed in Name Placeholders. The default is `%g/%a:%l`.	`fabric8.generator.name`
registry	A optional Docker registry used when doing Docker builds. It has no effect for OpenShift S2I builds.	`fabric8.generator.registry`

When used as properties they can be directly referenced with the property names above.

7.1.1. Java Applications

One of the most generic *Generators* is the `java-exec` generator. It is responsible for starting up arbitrary Java application. It knows how to deal with fat-jar applications where the application and all dependencies are included within a single jar and the `MANIFEST.MF` within the jar references a main class. But also flat classpath applications, where the dependencies are separate jar files and a main class is given.

If no main class is explicitly configured, the plugin first attempts to locate a fat jar. If the Maven build creates a JAR file with a `META-INF/MANIFEST.MF` containing a `Main-Class` entry, then this is considered to be the fat jar to use. If there are more than one of such files then the largest one is used.

If a main class is configured (see below) then the image configuration will contain the application jar plus all dependency jars. If no main class is configured as well as no fat jar being detected, then this *Generator* tries to detect a single main class by searching for `public static void main(String args[])` among the application classes. If exactly one class is found this is considered to be the main class. If no or more than one is found the *Generator* finally does nothing.

It will use the following base image by default, but as explained above and can be changed with the `from` configuration.

Table 24. Java Base Images

	Docker Build	S2I Build	ImageStream
Community	`fabric8/java-jboss-openjdk8-jdk`	`fabric8/s2i-java`	`fabric8-java`
Red Hat	`jboss-fuse-6/fis-java-openshift`	`jboss-fuse-6/fis-java-openshift`	`fis-java-openshift`

These images always refer to the latest tag. The *Red Hat* base images are selected, when the plugin itself is a Red Hat supported version (which is detected by the plugins version number).

When a `fromMode` of `istag` is used to specify an `ImageStreamTag` and when no `from` is given, then as default the `ImageStreamTag fis-java-openshift` in the namespace `openshift` is chosen. If you are using a RedHat variation of this plugin (i.e. if the version is ending with `-redhat`), then a `fromMode` of `istag` is the default, otherwise its `fromMode = "docker"` which use the a plain Docker image reference for the S2I builder image.

Beside the common configuration parameters described in the table common generator options the following additional configuration options are recognized:

Table 25. Java Application 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 to put the detected artefacts into. 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 given first a check is performed to detect a fat-jar (see above). Next a class is looked up by scanning `target/classes` for a single class with a main method. If no such class is found or if more than one is found, then this generator does nothing.	
prometheusPort	Port of the Prometheus jmx_exporter exposed by the base image. Set this to 0 if you don't want to expose the Prometheus port.	9779
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

The exposed ports are typically later on use by Enrichers to create default Kubernetes or OpenShift services.

You can add additional files to the target image within `baseDir` by placing files into `src/main/fabric8-includes`. These will be added with mode `0644`, while everything in `src/main/fabric8-includes/bin` will be added with `0755`.

7.1.2. Spring Boot

This generator is called `spring-boot` and gets activated when it finds a `spring-boot-maven-plugin` in the pom.xml.

This generator is based on the Java Application Generator and inherits all of its configuration values. The generated container port is read from the `server.port` property `application.properties`, defaulting to `8080` if it is not found. It also uses the same default images as the java-exec Generator.

Beside the common generator options and the java-exec options the following additional configuration is recognized:

Table 26. Spring-Boot configuration options

Element	Description	Default
color	If seth 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 `management.port` (for Spring Boot 1) or `management.server.port` (for Spring Boot 2) and `management.ssl.key-store` (for Spring Boot 1) or `management.server.ssl.key-store` (for Spring Boot 2) properties are set in `application.properties` otherwise or `server.ssl.key-store` property is set in `application.properties` then the probes are automatically set to use `https`.

The generator works differently when called together with `fabric8:watch`. In that case it enables support for Spring Boot Developer Tools which allows for hot reloading of the Spring Boot app. In particular, the following steps are performed:

- If a secret token is not provided within the Spring Boot application configuration `application.properties` or `application.yml` with the key `spring.devtools.remote.secret` then a custom secret token is created and added to `application.properties`
- Add `spring-boot-devtools.jar` as `BOOT-INF/lib/spring-devtools.jar` to the spring-boot fat jar.

Since during `fabric8:watch` the application itself within the `target/` directory is modified for allowing easy reloading you must ensure that you do a `mvn clean` before building an artefact which should be put into production. Since the released version are typically generated with a CI system which does a clean build anyway this should be only a theoretical problem.

7.1.3. Wildfly Swarm

The WildFly Swarm generator detects a WildFly Swarm build and disables the Prometheus Java agent because of this issue.

Otherwise this generator is identical to the java-exec generator. It supports the common generator options and the `java-exec` options.

7.1.4. Thorntail v2

The Thorntail v2 generator detects a Thorntail v2 build and disables the Prometheus Java agent because of this issue.

Otherwise this generator is identical to the java-exec generator. It supports the common generator options and the `java-exec` options.

7.1.5. Vert.x

The Vert.x generator detects an application using Eclipse Vert.x. It generates the metadata to start the application as a fat jar.

Currently, this generator is enabled if:

- you are using the Vert.x Maven Plugin (https://github.com/fabric8io/vertx-maven-plugin)

- you are depending on `io.vertx:vertx-core` and uses the Maven Shader plugin

Otherwise this generator is identical to the java-exec generator. It supports the common generator options and the `java-exec` options.

The generator automatically:

- enable metrics and JMX publishing of the metrics when `io.vertx:vertx-dropwizard-metrics` is in the project's classpath / dependencies.
- enable clustering when a Vert.x cluster manager is available in the project's classpath / dependencies. this is done by appending `-cluster` to the command line.
- Force IPv4 stack when `vertx-infinispan` is used.
- Disable the async DNS resolver to fallback to the regular JVM DNS resolver.

You can pass application parameter by setting the `JAVA_ARGS` env variable. You can pass system properties either using the same variable or using `JAVA_OPTIONS`. For instance, create `src/main/fabric8/deployment.yml` with the following content to configure `JAVA_ARGS`:

```
spec:
 template:
   spec:
     containers:
       - env:
         - name: JAVA_ARGS
           value: "-Dfoo=bar -cluster -instances=2"
```

7.1.6. Karaf

This generator named `karaf` kicks in when the build uses a `karaf-maven-plugin`. By default the following base images are used:

Table 27. Karaf Base Images

	Docker Build	S2I Build
Community	`fabric8/s2i-karaf`	`fabric8/s2i-karaf`
Red Hat	`jboss-fuse-6/fis-karaf-openshift`	`jboss-fuse-6/fis-karaf-openshift`

When a `fromMode` of `istag` is used to specify an `ImageStreamTag` and when no `from` is given, then as default the `ImageStreamTag` `fis-karaf-openshift:2.0` in the namespace `openshift` is chosen.

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

Table 28. Karaf configuration options

Element	Description	Default
baseDir	Directory within the generated image where to put the detected artefacts into. 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 given first a check is performed to detect a fat-jar (see above). Next a class is tried to be found by scanning `target/classes` for a single class with a main method. If no if found or more than one is found, then this generator does nothing.	
user	User and/or group under which the files should be added. The syntax of this options is descriped in Assembly Configuration.	`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

7.1.7. Web Applications

The `webapp` generator tries to detect WAR builds and selects a base servlet container image based on the configuration found in the `pom.xml`:

- A **Tomcat** base image is selected when a `tomcat6-maven-plugin` or `tomcat7-maven-plugin` is present or when a `META-INF/context.xml` could be found in the classes directory.
- A **Jetty** base image is selected when a `jetty-maven-plugin` is present or one of the files `WEB-INF/jetty-web.xml` or `WEB-INF/jetty-logging.properties` is found.
- A **Wildfly** base image is chosen for a given `jboss-as-maven-plugin` or `wildfly-maven-plugin` or when a Wildfly specific deployment descriptor like `jboss-web.xml` is found.

The base images chosen are:

Table 29. Webapp Base Images

	Docker Build	S2I Build
Tomcat	`fabric8/tomcat-8`	---
Jetty	`fabric8/jetty-9`	---
Wildfly	`jboss/wildfly`	---

S2I builds are currently not yet supported for the webapp generator.

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

Table 30. Webapp configuration options

Element	Description	Default
server	Fix server to use in the base image. Can be either **tomcat**, **jetty** or **wildfly**	

Element	Description	Default
targetDir	Where to put the war file into the target image. By default its selected by the base image chosen but can be overwritten with this option.	
user	User and/or group under which the files should be added. The syntax of this options is descriped in Assembly Configuration.	
path	Context path with which the application can be reached by default	/ (root context)
cmd	Command to use to start the container. By default the base images startup command is used.	
ports	Comma separated list of ports to expose in the image and which eventually are translated later to Kubernertes services. The ports depend on the base image and are selecte automatically. But they can be overwritten here.	

7.2. Generator API

The API is still a bit in flux and will be documented later. Please refer to the Generator Interface in the meantime.

Chapter 8. Enrichers

Enriching is the complementary concept to Generators. Whereas Generators are used to create and customize Docker images, Enrichers are use to create and customize Kubernetes and OpenShift resource objects.

There are a lot of similarities to Generators:

- Each Enricher has a unique name.
- Enrichers are looked up automatically from the plugin dependencies and there is a set of default enrichers delivered with this plugin.
- Enrichers are configured the same ways as generators

The Generator example is a good bluebprint, simply replace `<generator>` with `<enricher>`. The configuration is structural identical:

Table 31. Enricher configuration

Element	Description
`<includes>`	Contains one ore more `<include>` elements with enricher names which should be included. If given, only this list of enrichers are included in this order. The enrichers from every active profile are included, too. However the enrichers listed here are moved to the front of the list, so that they are called first. Use the profile `raw` if you want to explicitly set the complete list of enrichers.
`<excludes>`	Holds one or more `<exclude>` elements with enricher names to exclude. This means all the detected enrichers are used except the ones mentioned in this section.
`<config>`	Configuration for all enrichers. Each enricher supports a specific set of configuration values as described in its documentation. The subelements of this section are enricher names. E.g. for enricher `f8-service`, the sub-element is called `<f8-service>`. This element then holds the specific enricher configuration like `<name>` for the service name. Configuration coming from profiles are merged into this config, but not overriding the configuration specified here.

This plugin comes with a set of default enrichers. In addition custom enrichers can be easily added by providing implementation of the Enricher API and adding these as a dependency to the build.

8.1. Default Enrichers

fabric8-maven-plugin comes with a set of enrichers which are enabled by default. There are two categories of default enrichers:

- **Standard Enrichers** are used to add default resource object when they are missing or add common metadata extracted from the given build information
- **Fabric8 Enrichers** are specific to the fabric8 Microservice's platform. It adds icon annotations

which are visible in the fabric8 console or connections to the Continous Deliverys systems like Jenkins or Gogs.

- **OpenShift.io Enrichers** are specific to the OpenShift.io development environment. These enrichers can add metadata to resources to allow them to better integrate with OpenShift.io.

1. Default Enrichers Overview

Enricher	Description
f8-cd	Add CD metadata as annotations.
f8-doc-link	Add URL to documentation configured in the POM as annotation.
f8-grafana-link	Add a Grafana Dashboard URL as annotation.
f8-icon	Add an URL to icons for well known application types.
f8-prometheus	Add Prometheus annotations.
f8-maven-scm-enricher	Add Maven SCM information as annotations to the kubernetes/openshift resources
fmp-controller	Create default controller (replication controller, replica set or deployment) if missing.
fmp-dependency	Examine build dependencies for `kubernetes.yml` and add the objects found therein.
fmp-git	Check local `.git` directory and add build information as annotations.
fmp-ianaservice	Add a default portname looking up IANA services
fmp-image	Add the image name into a `PodSpec` of replication controller, replication sets and deployments, if missing.
fmp-name	Add a default name to every object which misses a name.
fmp-pod-annotation	Copy over annotations from a `Deployment` to a `Pod`
fmp-portname	Add a default portname for commonly known service.
fmp-project	Add Maven coordinates as labels to all objects.
fmp-service	Create a default service if missing and extrac ports from the Docker image configuration.
f8-maven-issue-mgmt	Add Maven Issue Management information as annotations to the kubernetes/openshift resources

Enricher	Description
fmp-revision-history	Add revision history limit (Kubernetes doc) as a deployment spec property to the Kubernetes/OpenShift resources.
fmp-triggers-annotation	Add ImageStreamTag change triggers on Kubernetes resources such as StatefulSets, ReplicaSets and DaemonSets using the `image.openshift.io/triggers` annotation.

8.1.1. Standard Enrichers

Default enrichers are used for adding missing resources or adding metadata to given resource objects. The following default enhancers are available out of the box

fmp-controller

fmp-service

This enricher is used to ensure that a service is present. This can be either directly configured with fragments or with the XML configuration, but it can be also automatically inferred by looking at the ports exposed by an image configuration. An explicit configuration always takes precedence over auto detection. For enriching an existing service this enricher actually works only on a configured service which matches with the configured (or inferred) service name.

The following configuration parameters can be used to influence the behaviour of this enricher:

Table 32. Default service enricher

Element	Description	Default
name	Service name to enrich by default. If not given here or configured elsewhere, the artifactId is used	
headless	whether a headless service without a port should be configured. A headless service has the `ClusterIP` set to `None` and will be only used if no ports are exposed by the image configuration or by the configuration `port`.	`false`
expose	If set to true, a label `expose` with value `true` is added which can be picked up by the fabric8 expose-controller to expose the service to the outside by various means. See the documentation of expose-controller for more details.	`false`
type	Kubernetes / OpenShift service type to set like *LoadBalancer*, *NodePort* or *ClusterIP*.	
port	The service port to use. By default the same port as the ports exposed in the image configuration is used, but can be changed with this parameter. See below for a detailed description of the format which can be put into this variable.	

Element	Description	Default
multiPort	Set this to `true` if you want all ports to be exposed from an image configuration. Otherwise only the first port is used as a service port.	`false`
protocol	Default protocol to use for the services. Must be `tcp` or `udp`	`tcp`
legacyPortMapping	If this mapping options is set to `true` then a pod exports ports 8080 or 9090 is mapped to a service port 80. This is deprecated and switched off by default. You can switch it on to get back the old behaviour or, even better, use `port` for setting the service port directly.	`false`

You specify the properties like for any enricher within the enrichers configuration like in

Example

```
<configuration>
  ..
  <enricher>
    <config>
      <fmp-service>
        <name>my-service</name>
        <type>NodePort</type>
        <multiPort>true</multiPort>
      </fmp-service>
    </config>
  </enricher>
</configuration>
```

Port specification

With the option `port` you can influence the mapping how ports are mapped from the pod to the service. By default and if this option is not given the ports exposed are dictated by the ports exposed from the Docker images contained in the pods. Remember, each image configured can be part of the pod. However you can expose also completely different ports as the images meta data declare.

The property `port` can contain a comma separated list of mappings of the following format:

```
<servicePort1>:<targetPort1>/<protocol>,<servicePort2>:<targetPort2>/<protocol>,....
```

where the `targetPort` and `<protocol>` specification is optional. These ports are overlayed over the ports exposed by the images, in the given order.

This is best explained by some examples.

For example if you have a pod which exposes a Microservice on port 8080 and you want to expose it as a service on port 80 (so that it can be accessed with `http://myservice`) you can simply use the following enricher configuration:

Example

```
<configuration>
  <enricher>
    <config>
      <fmp-service>
        <name>myservice</name>
        <port>80:8080</port> ①
      </fmp-service>
    </config>
  </enricher>
</configuration>
```

① 80 is the service port, 8080 the port opened in from the pod's images

If your pod *exposes* their ports (which e.g. all generator do), then you can even omit the 8080 here (i.e. `<port>80</port>`). In this case the *first* port exposed will be mapped to port 80, all other exposed ports will be omitted.

By default an automatically generated service only exposes the first port, even when more ports are exposed. When you want to map multiple ports you need to set the config option `<multiPort>true</multiPort>`. In this case you can also provide multiple mappings as a comma separated list in the `<port>` specification where each element of the list are the mapping for the first, second, ... port.

A more (and bit artificially constructed) specification could be `<port>80,9779:9779/udp,443</port>`. Assuming that the image exposes ports `8080` and `8778` (either directly or via generators) and we have switched on multiport mode, then the following service port mappings will be performed for the automatically generated service:

- Pod port 8080 is mapped to service port 80.
- Pod port 9779 is mapped to service port 9779 with protocol UDP. Note how this second entry overrides the pod exposed port 8778.
- Pod port 443 is mapped to service port 443.

This example show also the mapping rules:

- Port specification in `port` always override the port meta data of the contained Docker images (i.e. the ports exposed)
- You can always provide a complete mapping with `port` on your own
- The ports exposed by the images serve as *default values* which are used if not specified by this configuration option.
- You can map ports which are *not* exposed by the images by specifying them as target ports.

Multiple ports are **only** mapped when *multiPort* mode is enabled (which is switched off by default). If *multiPort* mode is disabled, only the first port from the list of mapped ports as calculated like above is taken.

When you set `legacyPortMapping` to true than ports 8080 to 9090 are mapped to port 80 automatically if not explicitly mapped via `port`. I.e. when an image exposes port 8080 with a legacy mapping this mapped to a service port 80, not 8080. You *should not* switch this on for any good reason. In fact it might be that this option can vanish anytime.

fmp-image

fmp-name

fmp-portname

fmp-pod-annotation

fmp-ianaservice

fmp-project

Enricher that adds standard labels and selectors to generated resources (e.g. `app`, `group`, `provider`, `version`).

The `fmp-project` enricher supports the following configuration options:

Option	Description	Default
`useProjectLabel`	Enable this flag to turn on the generation of the old `project` label in Kubernetes resources. The `project` label has been replaced by the `app` label in newer versions of the plugin.	`false`

fmp-git

fmp-dependency

fmp-volume-permission

Enricher which fixes the permission of persistent volume mount with the help of an init container.

fmp-autotls

Enricher which adds appropriate annotations and volumes to enable OpenShift's automatic Service Serving Certificate Secrets. This enricher adds an init container to convert the service serving certificates from PEM (the format that OpenShift generates them in) to a JKS-format Java keystore ready for consumption in Java services.

This enricher is disabled by default. In order to use it, you must configure the Fabric8 Maven plugin to use this enricher:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.3.0</version>
  <executions>
    <execution>
      <goals>
        <goal>resource</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <enricher>
      <includes>
        <include>fmp-autotls</include>
      </includes>
      <config>
        <fmp-autotls>
          ...
        </fmp-autotls>
      </config>
    </enricher>
  </configuration>
</plugin>
```

The auto-TLS enricher supports the following configuration options:

Option	Description	Default
tlsSecretName	The name of the secret to be used to store the generated service serving certs.	<project.artifactId>-tls
tlsSecretVolumeMountPoint	Where the service serving secret should be mounted to in the pod.	/var/run/secrets/fabric8.io/tls-pem
tlsSecretVolumeName	The name of the secret volume.	tls-pem
jksVolumeMountPoint	Where the generated keystore volume should be mounted to in the pod.	/var/run/secrets/fabric8.io/tls-jks
jksVolumeName	The name of the keystore volume.	tls-jks
pemToJKSInitContainerImage	The name of the image used as an init container to convert PEM certificate/key to Java keystore.	jimmidyson/pemtokeystore:v0.1.0
pemToJKSInitContainerName	the name of the init container to convert PEM certificate/key to Java keystore.	tls-jks-converter
keystoreFileName	The name of the generated keystore file.	keystore.jks
keystorePassword	The password to use for the generated keystore.	changeit

Option	Description	Default
`keystoreCertAlias`	The alias in the keystore used for the imported service serving certificate.	`server`

8.1.2. Fabric8 Enrichers

Fabric8 enrichers are used for providing the connection to other components of the fabric8 Microservices platform. They are useful to add icons to to application or links to documentation sites.

f8-cd

f8-doc-link

f8-grafana-link

f8-icon

f8-karaf-health-check

This enricher adds kubernetes readiness and liveness probes with Apache Karaf. This requires that `fabric8-karaf-checks` has been enabled in the Karaf startup features.

The enricher will use the following settings by default:

- port = `8181`
- scheme = `HTTP`

and use paths `/readiness-check` for readiness check and `/health-check` for liveness check.

These options cannot be configured.

f8-prometheus

This enricher adds Prometheus annotation like:

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      prometheus.io/scrape: "true"
      prometheus.io/port: 9779
```

By default the enricher inspects the images' BuildConfiguration and add the annotations if the port 9779 is listed. You can force the plugin to add annotations by setting enricher's config

`prometheusPort`

f8-spring-boot-health-check

This enricher adds kubernetes readiness and liveness probes with Spring Boot. This requires the following dependency has been enabled in Spring Boot

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

The enricher will try to discover the settings from the `application.properties` / `application.yaml` Spring Boot configuration file.

The port number is read from the `management.port` option, and will use the default value of `8080` The scheme will use HTTPS if `server.ssl.key-store` option is in use, and fallback to use `HTTP` otherwise.

The enricher will use the following settings by default:

- `readinessProbeInitialDelaySeconds` : `10`
- `readinessProbePeriodSeconds` : *<kubernetes-default>*
- `livenessProbeInitialDelaySeconds` : `180`
- `livenessProbePeriodSeconds` : *<kubernetes-default>*
- `timeoutSeconds` : *<kubernetes-default>*

These values can be configured by the enricher in the `fabric8-maven-plugin` configuration as shown below:

```
      <plugin>
        <groupId>io.fabric8</groupId>
        <artifactId>fabric8-maven-plugin</artifactId>
        <version>3.3.0</version>
        <executions>
          <execution>
            <id>fmp</id>
            <goals>
              <goal>resource</goal>
              <goal>helm</goal>
              <goal>build</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <enricher>
            <config>
              <spring-boot-health-check>
                <timeoutSeconds>5</timeoutSeconds>
                <readinessProbeInitialDelaySeconds>
30</readinessProbeInitialDelaySeconds>
              </spring-boot-health-check>
            </config>
          </enricher>
        </configuration>
      </plugin>
```

f8-wildfly-swarm-health-check

This enricher adds kubernetes readiness and liveness probes with WildFly Swarm. This requires the following fraction has been enabled in WildFly Swarm

```
    <dependency>
      <groupId>org.wildfly.swarm</groupId>
      <artifactId>microprofile-health</artifactId>
    </dependency>
```

The enricher will use the following settings by default:

- port = `8080`
- scheme = `HTTP`
- path = `/health`

These values can be configured by the enricher in the `fabric8-maven-plugin` configuration as shown below:

```
    <plugin>
      <groupId>io.fabric8</groupId>
      <artifactId>fabric8-maven-plugin</artifactId>
      <version>3.3.0</version>
      <executions>
        <execution>
          <id>fmp</id>
          <goals>
            <goal>resource</goal>
            <goal>helm</goal>
            <goal>build</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <enricher>
          <config>
            <wildfly-swarm-health-check>
              <port>4444</port>
              <scheme>HTTPS</scheme>
              <path>health/myapp</path>
            </wildfly-swarm-health-check>
          </config>
        </enricher>
      </configuration>
    </plugin>
```

f8-thorntail-v2-health-check

This enricher adds kubernetes readiness and liveness probes with Thorntail v2. This requires the following fraction has been enabled in Thorntail

```
  <dependency>
    <groupId>io.thorntail</groupId>
    <artifactId>microprofile-health</artifactId>
  </dependency>
```

The enricher will use the following settings by default:

- port = `8080`
- scheme = `HTTP`
- path = `/health`

These values can be configured by the enricher in the `fabric8-maven-plugin` configuration as shown below:

```
      <plugin>
        <groupId>io.fabric8</groupId>
        <artifactId>fabric8-maven-plugin</artifactId>
        <version>3.3.0</version>
        <executions>
          <execution>
            <id>fmp</id>
            <goals>
              <goal>resource</goal>
              <goal>helm</goal>
              <goal>build</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <enricher>
            <config>
              <thorntail-v2-health-check>
                <port>4444</port>
                <scheme>HTTPS</scheme>
                <path>health/myapp</path>
              </thorntail-v2-health-check>
            </config>
          </enricher>
        </configuration>
      </plugin>
```

f8-vertx-health-check

This enricher adds kubernetes readiness and liveness probes with Eclipse Vert.x applications. The readiness probe lets Kubernetes detects when the application is ready, while the liveness probe allows Kubernetes to verify that the application is still alive.

This enricher allows configuring the readiness and liveness probes. Are supported: `http` (emit HTTP requests), `tcp` (open a socket), `exec` (execute a command).

By default, this enricher uses the same configuration for liveness and readiness probes. But specific configurations can be provided. The configurations can be overridden using project's properties.

Using the f8-vertx-health-check enricher

The enricher is automatically executed if your project uses the `vertx-maven-plugin` or depends on `io.vertx:vertx-core`. However, by default, no health check will be added to your deployment.

Minimal configuration

The minimal configuration to add health checks is the following:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.41</version>
  <executions>
    <execution>
      <id>fmp</id>
      <goals>
        <goal>resource</goal>
        <goal>helm</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <enricher>
      <config>
        <vertx-health-check>
            <path>/health</path>
        </vertx-health-check>
      </config>
    </enricher>
  </configuration>
</plugin>
```

It configures the readiness and liveness health checks using HTTP requests on the port `8080` (default port) and on the path `/health`. The defaults are:

- port = `8080` (for HTTP)
- scheme = `HTTP`
- path = *none* (disabled)

the previous configuration can also be given use project's properties:

```
<properties>
    <vertx.health.path>/health</vertx.health.path>
</properties>
```

Configuring differently the readiness and liveness health checks

You can provide two different configuration for the readiness and liveness checks:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.41</version>
  <executions>
    <execution>
      <id>fmp</id>
      <goals>
        <goal>resource</goal>
        <goal>helm</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <enricher>
      <config>
        <vertx-health-check>
            <readiness>
              <path>/ready</path>
            </readiness>
            <liveness>
              <path>/health</path>
            </liveness>
        </vertx-health-check>
      </config>
    </enricher>
  </configuration>
</plugin>
```

You can also use the `readiness` and `liveness` chunks in user properties:

```
<properties>
    <vertx.health.readiness.path>/ready</vertx.health.readiness.path>
    <vertx.health.liveness.path>/ready</vertx.health.liveness.path>
</properties>
```

Shared (generic) configuration can be set outside of the specific configuration. For instance, to use the port 8081:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.41</version>
  <executions>
    <execution>
      <id>fmp</id>
      <goals>
        <goal>resource</goal>
        <goal>helm</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <enricher>
      <config>
        <vertx-health-check>
            <port>8081</port>
            <readiness>
              <path>/ready</path>
            </readiness>
            <liveness>
              <path>/health</path>
            </liveness>
        </vertx-health-check>
      </config>
    </enricher>
  </configuration>
</plugin>
```

Or:

```
<properties>
    <vertx.health.port>8081</vertx.health.port>
    <vertx.health.readiness.path>/ready</vertx.health.readiness.path>
    <vertx.health.liveness.path>/ready</vertx.health.liveness.path>
</properties>
```

Configuration Structure

The configuration is structured as follows

```
<config>
    <vertx-health-check>
        <!-- Generic configuration, applied to both liveness and readiness -->
        <path>/both</path>
        <liveness>
            <!-- Specific configuration for the liveness probe -->
            <port-name>ping</port-name>
        </liveness>
        <readiness>
            <!-- Specific configuration for the readiness probe -->
            <port-name>ready</port-name>
        </readiness>
    </vertx-health-check>
</config>
```

The same structured is used in project's properties:

```
<!-- Generic configuration given as vertx.health.$attribute -->
<vertx.health.path>/both</vertx.health.path>
<!-- Specific liveness configuration given as vertx.health.liveness.$attribute -->
<vertx.health.liveness.port-name>ping</vertx.health.liveness.port-name>
<!-- Specific readiness configuration given as vertx.health.readiness.$attribute -->
<vertx.health.readiness.port-name>ready</vertx.health.readiness.port-name>
```

Important: Project's properties override the configuration provided in the plugin configuration. The overriding rules are: *specific properties* > *generic properties* > *specific configuration* > *generic configuration*.

Probe configuration

You can configure the different aspect of the probes. These attributes can be configured for both the readiness and liveness probes or be specific to one.

Table 33. Probe configuration

Name	Description
`type`	The probe type among `http` (default), `tcp` and `exec`.
`initial-delay`	Number of seconds after the container has started before probes are initiated.
`period`	How often (in seconds) to perform the probe.
`timeout`	Number of seconds after which the probe times out.
`success-threshold`	Minimum consecutive successes for the probe to be considered successful after having failed.

Name	Description
`failure-threshold`	Minimum consecutive failures for the probe to be considered failed after having succeeded.

More details about probes are available on https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/.

HTTP probe configuration

When using HTTP `GET` requests to determine readiness or liveness, several aspects can be configured. HTTP probes are used by default. To be more specific set the `type` attribute to `http`.

Table 34. HTTP probes configuration

Name	Description	Default
`scheme`	Scheme to use for connecting to the host.	`HTTP`
`path`	Path to access on the HTTP server. An empty path disable the check	
`headers`	Custom headers to set in the request. HTTP allows repeated headers. Cannot be configured using project's properties. An example is available below.	
`port`	Number of the port to access on the container. A 0 or negative number disable the check.	8080
`port-name`	Name of the port to access on the container. If neither the `port` or the `port-name` is set, the check is disabled. If both are set the configuration is considered invalid.	

Here is an example of HTTP probe configuration:

```
<config>
    <vertx-health-check>
        <initialDelay>3</initialDelay>
        <period>3</period>
        <liveness>
            <port>8081</port>
            <path>/ping</path>
            <scheme>HTTPS</scheme>
            <headers>
                <X-Custom-Header>Awesome</X-Custom-Header>
            </headers>
        </liveness>
        <readiness>
            <!-- disable the readiness probe -->
            <port>-1</port>
        </readiness>
    </vertx-health-check>
</config>
```

TCP probe configuration

You can also configure the probes to just open a socket on a specific port. The `type` attribute must be set to `tcp`.

Table 35. TCP probes configuration

Name	Description
`port`	Number of the port to access on the container. A 0 or negative number disable the check.
`port-name`	Name of the port to access on the container. If neither the `port` or the `port-name` is set, the check is disabled. If both are set the configuration is considered invalid.

For example:

```
<config>
    <vertx-health-check>
        <initialDelay>3</initialDelay>
        <period>3</period>
        <liveness>
            <type>tcp</type>
            <port>8081</port>
        </liveness>
        <readiness>
            <-- use HTTP Get probe -->
            <path>/ping</path>
            <port>8080</port>
        </readiness>
    </vertx-health-check>
</config>
```

Exec probe configuration

You can also configure the probes to execute a command. If the command succeeds, it returns 0, and Kubernetes consider the pod to be alive and healthy. If the command returns a non-zero value, Kubernetes kills the pod and restarts it. To use a command, you must set the `type` attribute to `exec`:

```
<config>
    <vertx-health-check>
        <initialDelay>3</initialDelay>
        <period>3</period>
        <liveness>
            <type>exec</type>
            <command>
                <cmd>cat</cmd>
                <cmd>/tmp/healthy</cmd>
            </command>
        </liveness>
        <readiness>
            <-- use HTTP Get probe -->
            <path>/ping</path>
            <port>8080</port>
        </readiness>
    </vertx-health-check>
</config>
```

As you can see in the snippet above the command is passed using the `command` attribute. This attribute cannot be configured using project's properties. An empty command disables the check.

Disabling health checks

You can disables the checks by setting:

- the `port` to 0 or to a negative number for `http` and `tcp` probes

- the `command` to an empty list for `exec`

In the first case, you can use project's properties to disable them:

```
<!-- Disables <code>tcp</code> and <code>http</code> probes -->
<vertx.health.port>-1</vertx.health.port>
```

For `http` probes, an empty or not set `path` also disable the probe.

f8-maven-scm-enricher

This enricher adds additional SCM related metadata to all objects supporting annotations. These metadata will be added only if SCM information is present in the maven `pom.xml` of the project.

The following annotations will be added to the objects that supports annotations,

Table 36. Maven SCM Enrichers Annotation Mapping

Maven SCM Info	Annotation	Description
scm/connection	fabric8.io/scm-con-url	The SCM connection that will be used to connect to the project's SCM
scm/developerConnection	fabric8.io/scm-devcon-url	The SCM Developer Connection that will be used to connect to the project's developer SCM
scm/tag	fabric8.io/scm-tag	The SCM tag that will be used to checkout the sources, like HEAD dev-branch etc.,
scm/url	fabric8.io/scm-url	The SCM web url that can be used to browse the SCM over web browser

Lets say you have a maven pom.xml with the following scm information,

```
<scm>
    <connection>scm:git:git://github.com/fabric8io/fabric8-maven-
plugin.git</connection>
    <developerConnection>scm:git:git://github.com/fabric8io/fabric8-maven-
plugin.git</developerConnection>
    <url>git://github.com/fabric8io/fabric8-maven-plugin.git</url>
</scm>
```

This infomation will be enriched as annotations in the generated manifest like,

```
...
  kind: Service
  metadata:
    annotations
      fabric8.io/scm-con-url: "scm:git:git://github.com/fabric8io/fabric8-maven-
plugin.git"
      fabric8.io/scm-devcon-url: "scm:git:git://github.com/fabric8io/fabric8-maven-
plugin.git"
      fabric8.io/scm-tag: "HEAD"
      fabric8.io/scm-url: "git://github.com/fabric8io/fabric8-maven-plugin.git"
...
```

f8-maven-issue-mgmt

This enricher adds additional Issue Management related metadata to all objects supporting annotations. These metadata will be added only if the Issue Management information in available in maven `pom.xml` of the project.

The following annotations will be added to the objects that supports annotations,

Table 37. Maven Issue Tracker Enrichers Annotation Mapping

Maven Issue Tracker Info	Annotation	Description
issueManagement/system	fabric8.io/issue-system	The Issue Management system like Bugzilla, JIRA, GitHub etc.,
issueManagement/url	fabric8.io/issue-tracker-url	The Issue Management url e.g. GitHub Issues Url

Lets say you have a maven pom.xml with the following issue management information,

```
<issueManagement>
   <system>GitHub</system>
   <url>https://github.com/fabric8io/vertx-maven-plugin/issues/</url>
</issueManagement>
```

This information will be enriched as annotations in the generated manifest like,

```
...
  kind: Service
  metadata:
    annotations:
      fabric8.io/issue-system: "GitHub"
      fabric8.io/issue-tracker-url: "https://github.com/fabric8io/vertx-maven-
plugin/issues/"
...
```

fmp-revision-history

This enricher adds `spec.revisionHistoryLimit` property to deployment spec of Kubernetes/OpenShift resources. A deployment's revision history is stored in the replica sets, that specifies the number of old ReplicaSets to retain in order to allow rollback. For more information read Kubernetes documentation.

The following configuration parameters can be used to influence the behaviour of this enricher:

Table 38. Default revision history enricher

Element	Description	Default
limit	Number of revision histories to retain	2

Just as any other enricher you can specify required properties with in the enricher's configuration as below,

```
...
<enricher>
    <config>
        <fmp-revision-history>
            <limit>8</limit>
        </fmp-revision-history>
    </config>
</enricher>
...
```

This information will be enriched as spec property in the generated manifest like,

```
...
kind: Deployment
spec:
  revisionHistoryLimit: 8
...
```

8.1.3. OpenShift.io Enrichers

OpenShift.io enrichers add metadata to resources in order to better integrate them into OpenShift.io. They are not enabled by default, but can be activated with the `osio` profile.

osio-space-label

This enricher adds a label named `space` to all resources generated by the `fabric8:resource` goal. The value of this label identifies the OpenShift.io space that this application belongs to. For more information about spaces, see the OpenShift.io documentation.

To provide the value of the `space` label to be applied by this enricher, use one of the following:

1. Set the system property `fabric8.enricher.osio-space-label.space` when building:

```
mvn -Dfabric8.enricher.osio-space-label.space=mySpace fabric8:resource
```

2. Define property `fabric8.enricher.osio-space-label.space` in your project's pom.xml:

```
...
  <properties>
    <fabric8.enricher.osio-space-label.space>
      mySpace
    </fabric8.enricher.osio-space-label.space>
  </properties>
...
```

3. Specify in the configuration for `fabric8-maven-plugin`:

```
...
  <plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    ...
    <configuration>
      <enricher>
        <config>
          <osio-space-label>
            <space>mySpace</space>
          </osio-space-label>
        </config>
      </enricher>
    </configuration>
  </plugin>
...
```

Regardless of which above method you choose to configure the enricher, the resulting resources will have the `space` label applied, like the following:

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  ...
  labels:
    space: mySpace
    app: myproject
    provider: fabric8
    version: 1.0.0-SNAPSHOT
    group: com.example
  ...
```

fmp-triggers-annotation

This enricher adds ImageStreamTag change triggers on Kubernetes resources that support the `image.openshift.io/triggers` annotation, such as StatefulSets, ReplicaSets and DaemonSets.

The trigger is added to all containers that apply, but can be restricted to a limited set of containers using the following configuration:

```
...
<enricher>
    <config>
        <fmp-triggers-annotation>
            <containers>container-name-1,c2</containers>
        </fmp-triggers-annotation>
    </config>
</enricher>
...
```

8.2. Enricher API

howto write your own enricher and install them

Chapter 9. Profiles

Profiles can be used to combine a set of enrichers and generators and to give this combination a referable name.

Profiles are defined in YAML. The following example shows a simple profiles which uses only the Spring Boot generator and some enrichers adding for adding default resources:

Profile Definition

```
- name: my-spring-boot-apps ①
  generator: ②
    includes:
      - spring-boot
  enricher: ③
    includes: ④
      # Default Deployment object
      - fmp-controller
      # Add a default service
      - fmp-service
    excludes: ⑤
      - f8-icon
    config: ⑥
      fmp-service:
        # Expose service as NodePort
        type: NodePort
  order: 10 ⑦
- name: another-profile
  ....
```

① Profile's name

② Generators to use

③ Enrichers to use

④ List of enricher to **include** in that given order

⑤ List of enricher to **exclude**

⑥ Configuration for services an enrichers

⑦ An order which influences the way how profiles with the same name are merged

Each `profiles.yml` has a list of profiles which are defined with these elements:

Table 39. Profile elements

Element	Description
name	Profile name. This plugin comes with a set of predefined profiles. Those profiles can be extended by defining a custom profile with the same name of the profile to extend.

Element	Description
generator	List of generator definitions. See below for the format of this definitions.
enricher	List of enrichers definitions. See below for the format of this definitions.
order	The order of the profile which is used when profiles of the same name are merged.

9.1. Generator and Enricher definitions

The definition of generators and enrichers in the profile follow the same format:

Table 40. Generator and Enericher definition

Element	Description
includes	List of generators or enrichers to include. The order in the list determines the order in which the processors are applied.
excludes	List of generators or enrichers. These have precedences over *includes* and will exclude a processor even when referenced in an *includes* sections
config	Configuration for genertors or enrichers. This is a map where the keys are the name of the processor to configure and the value is again a map with configuration keys and values specific to the processor. See the documentation of the respective generator or enricher for the available configuration keys.

9.2. Lookup order

Profiles can be defined externally either directly as a build resource in `src/main/fabric8/profiles.yml` or provided as part of a plugin's dependency where it is supposed to be included as `META-INF/fabric8/profiles.yml`. Multiple profiles can be include in these `profiles.yml` descriptors as a list:

If a profile is used then it is looked up from various places in the following order:

- From the compile and plugin classpath from `META-INF/fabric8/profiles-default.yml`. These files are reserved for profiles defined by this plugin
- From the compile and plugin classpath from `META-INF/fabric8/profiles.yml`. Use this location for defining your custom profiles which you want to include via dependencies.
- From the project in `src/main/fabric8/profiles.yml`. The directory can be tuned with the plugin option `resourceDir` (property: `fabric8.resourceDir`)

When multiple profiles of the same name are found, then these profiles are merged. If profile have an order number, then the *higher* order takes precedences when merging profiles.

For *includes* of the same processors, the processor is moved to the earliest position. E.g consider the following two profiles with the name `my-profile`

Profile A

```
name: my-profile
enricher:
  includes: [ e1, e2 ]
```

Profile B

```
name: my-profile
enricher:
  includes: [ e3, e1 ]
order: 10
```

then when merged results in the following profile (when no order is given, it defaults to 0):

Profile merged

```
name: my-profile
enricher:
  includes: [ e1, e2, e3 ]
order: 10
```

Profile with the same order number are merged according to the lookup order described above, where the latter profile is supposed to have a higher order.

The configuration for enrichers and generators are merged, too, where higher order profiles override configuration values with the same key of lower order profile configuration.

9.3. Using Profiles

Profiles can be selected by defining them in the plugin configuration, by giving a system property or by using special directories in the directory holding the resource fragments.

Profile used in plugin configuration

Here is an example how the profile can be used in a plugin configuration:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <configuration>
    <profile>my-spring-boot-apps</profile> ①
    .....
  </configuration>
</plugin>
```

① Name which select the profile from the `profiles.yml`

Profile as system property

Alternatively a profile can be also specified on the command line when calling Maven:

```
mvn -Dfabric8.profile=my-spring-boot-apps fabric8:build fabric8:deploy
```

If a configuration for enrichers and generators are provided as part of the plugin's `<configuration>` then this takes precedence over any profile specified.

Profiles for resource fragments

Profiles are also very useful when used together with resource fragments in `src/main/fabric8`. By default the resource objects defined here are enriched with the configured profile (if any). A different profile can be selected easily by using a sub directory within `src/main/fabric8`. The name of each sub directory is interpreted as a profile name and all resource definition files found in this sub directory are enhanced with the enhancers defined in this profile.

For example, consider the following directory layout:

```
src/main/fabric8:
  app-rc.yml
  app-svc.yml
  raw/ -->
    couchbase-rc.yml
    couchbase-svc.yml
```

Here, the resource descriptors `app-rc.yml` and `app-svc.yml` are enhanced with the enrichers defined in the main configuration. The files two files `couchbase-rc.yml` and `couchbase-svc.yml` in the sub directory `raw/` instead are enriched with the profile **raw**. This is a predefined profile which includes no enricher at all, so the couchbase resource objects are not enriched and taken over literally. This is an easy way how you can fine tune enrichment for different object set.

9.4. Predefined Profiles

This plugin comes with a list of the following predefined profiles:

Table 41. Predefined Profiles

Profile	Description
default	The default profile which is active if no profile is specified. It consists of a curated set of generator and enrichers. See below for the current definition.
minimal	This profile contains no generators and only enrichers for adding default objects (controller and services). No other enrichment is included.
explicit	Like default but without adding default objects like controllers and services.
aggregate	Includes no generators and only the fmp-dependency enricher for picking up and combining resources from the compile time dependencies.

Profile	Description
internal-microservice	Do not expose a port for the service to generate. Otherwise the same as the *default* profile.
osio	Includes everything in the *default* profile, plus additional enrichers and generators relevant only to OpenShift.io.

9.5. Extending Profiles

A profile can also extend another profile to avoid repetition e.g of generators if the profile is only about including certain enrichers. For example, for a profile like:

```
- name: minimal
   extends: default
   enricher:
     includes:
     - fmp-name
     - fmp-controller
     - fmp-service
     - fmp-image
     - fmp-project
     - fmp-debug
```

one then would not need to repeat all generators as they are inherited from the `default` profile.

Default Profile

```
# Default profile which is always activated
- name: default
  enricher:
    # The order given in "includes" is the order in which enrichers are called
    includes:
    - fmp-name
    - fmp-controller
    - fmp-service
    - fmp-image
    - fmp-portname
    - fmp-ianaservice
    - fmp-project
    - fmp-dependency
    - fmp-pod-annotations
    - fmp-git
    # TODO: Documents and verify enrichers below
    - fmp-debug
    - fmp-merge
    - fmp-remove-build-annotations
    - fmp-volume-permission
    # ----------------------------------------
```

```
    - f8-cd
    - f8-cd-doc-link
    - f8-cd-grafana-link
    - f8-icon
    # TODO: Document and verify enrichers below
    - f8-expose
    # Health checks
    - fmp-openshift-route
    - spring-boot-health-check
    - wildfly-swarm-health-check
    - thorntail-v2-health-check
    - karaf-health-check
    - vertx-health-check
    - docker-health-check
    - f8-prometheus
    - f8-maven-scm
    - f8-maven-issue-mgmt
    # Dependencies shouldn't be enriched anymore, therefor it's last in the list
    - fmp-dependency
    - f8-watch
    - fmp-revision-history
    - fmp-docker-registry-secret
    - fmp-triggers-annotation

  generator:
    # The order given in "includes" is the order in which generators are called
    includes:
    - spring-boot
    - wildfly-swarm
    - thorntail-v2
    - karaf
    - vertx
    - java-exec
    - webapp
  watcher:
    includes:
    - spring-boot
    - docker-image
```

Chapter 10. Access configuration

10.1. Docker Access

This section is work-in-progress and not yet finished

For Kubernetes builds the fabric8-maven-plugin uses the Docker remote API so the URL of your Docker Daemon must be specified. The URL can be specified by the dockerHost or machine configuration, or by the `DOCKER_HOST` environment variable. If not given

The Docker remote API supports communication via SSL and authentication with certificates. The path to the certificates can be specified by the certPath or machine configuration, or by the `DOCKER_CERT_PATH` environment variable.

10.2. OpenShift and Kubernetes Access

If no `DOCKER_HOST` is set and no unix socket could be accessed under `/var/run/docker.sock` then f-m-p checks whethe gofabric8 is in the path and uses `gofabric8 docker-env` to get the connection parameter to the Docker host exposed by

Chapter 11. Registry handling

Docker uses registries to store images. The registry is typically specified as part of the name. I.e. if the first part (everything before the first `/`) contains a dot (`.`) or colon (`:`) this part is interpreted as an address (with an optionally port) of a remote registry. This registry (or the default `docker.io` if no registry is given) is used during push and pull operations. This plugin follows the same semantics, so if an image name is specified with a registry part, this registry is contacted. Authentication is explained in the next section.

There are some situations however where you want to have more flexibility for specifying a remote registry. This might be, because you do not want to hard code a registry into `pom.xml` but provide it from the outside with an environment variable or a system property.

This plugin supports various ways of specifying a registry:

- If the image name contains a registry part, this registry is used unconditionally and can not be overwritten from the outside.
- If an image name doesn't contain a registry, then by default the default Docker registry `docker.io` is used for push and pull operations. But this can be overwritten through various means:
  - If the `<image>` configuration contains a `<registry>` subelement this registry is used.
  - Otherwise, a global configuration element `<registry>` is evaluated which can be also provided as system property via `-Ddocker.registry`.
  - Finally an environment variable `DOCKER_REGISTRY` is looked up for detecting a registry.

This registry is used for pulling (i.e. for autopull the base image when doing a `fabric8:build`) and pushing with `fabric8:push`. However, when these two goals a are combined on the command line like in `mvn -Ddocker.registry=myregistry:5000 package fabric8:build fabric8:push` the same registry is used for both operation. For a more fine grained control, separate registries for *pull* and *push* can be specified.

- In the plugin's configuration with the parameters `<pullRegistry>` and `<pushRegistry>`, respectively.
- With the system properties `docker.pull.registry` and `docker.push.registry`, respectively.

Example

```
<configuration>
  <registry>docker.jolokia.org:443</registry>
  <images>
    <image>
      <!-- Without an explicit registry ... -->
      <name>jolokia/jolokia-java</name>
      <!-- ... hence use this registry -->
      <registry>docker.ro14nd.de</registry>
      ....
    <image>
    <image>
      <name>postgresql</name>
      <!-- No registry in the name, hence use the globally
           configured docker.jolokia.org:443 as registry -->
      ....
    </image>
    <image>
      <!-- Explicitely specified always wins -->
      <name>docker.example.com:5000/another/server</name>
    </image>
  </images>
</configuration>
```

There is some special behaviour when using an externally provided registry like described above:

- When *pulling*, the image pulled will be also tagged with a repository name **without** registry. The reasoning behind this is that this image then can be referenced also by the configuration when the registry is not specified anymore explicitly.
- When *pushing* a local image, temporarily a tag including the registry is added and removed after the push. This is required because Docker can only push registry-named images.

Chapter 12. Authentication

When pulling (via the `autoPull` mode of `fabric8:start`) or pushing image, it might be necessary to authenticate against a Docker registry.

There are five different locations searched for credentials. In order, these are:

- Providing system properties `docker.username` and `docker.password` from the outside.
- Using a `<authConfig>` section in the plugin configuration with `<username>` and `<password>` elements.
- Using OpenShift configuration in `~/.config/kube`
- Using a `<server>` configuration in `~/.m2/settings.xml`
- Login into a registry with `docker login` (credentials in `~/.docker/config.json`)

Using the username and password directly in the `pom.xml` is not recommended since this is widely visible. This is most easiest and transparent way, though. Using an `<authConfig>` is straight forward:

```
<plugin>
  <configuration>
    <image>consol/tomcat-7.0</image>
    ...
    <authConfig>
      <username>jolokia</username>
      <password>s!cr!t</password>
    </authConfig>
  </configuration>
</plugin>
```

The system property provided credentials are a good compromise when using CI servers like Jenkins. You simply provide the credentials from the outside:

Example

```
mvn -Ddocker.username=jolokia -Ddocker.password=s!cr!t fabric8:push
```

The most secure and also the most *mavenish* way is to add a server to the Maven settings file `~/.m2/settings.xml`:

Example

```
<servers>
  <server>
    <id>docker.io</id>
    <username>jolokia</username>
    <password>s!cr!t</password>
  </server>
  ....
</servers>
```

The server id must specify the registry to push to/pull from, which by default is central index `docker.io` (or `index.docker.io` / `registry.hub.docker.com` as fallbacks). Here you should add your docker.io account for your repositories. If you have multiple accounts for the same registry, the second user can be specified as part of the ID. In the example above, if you have a second account 'fabric8io' then use an `<id>docker.io/fabric8io</id>` for this second entry. I.e. add the username with a slash to the id name. The default without username is only taken if no server entry with a username appended id is chosen.

As a final fallback, this plugin consults `~/.docker/config.json` for getting to the credentials. Within this file credentials are stored when connecting to a registry with the command `docker login` from the command line.

12.1. Pull vs. Push Authentication

The credentials lookup described above is valid for both push and pull operations. In order to narrow things down, credentials can be be provided for pull or push operations alone:

In an `<authConfig>` section a sub-section `<pull>` and/or `<push>` can be added. In the example below the credentials provider are only used for image push operations:

Example

```
<plugin>
  <configuration>
    <image>consol/tomcat-7.0</image>
    ...
    <authConfig>
      <push>
         <username>jolokia</username>
         <password>s!cr!t</password>
      </push>
    </authConfig>
  </configuration>
</plugin>
```

When the credentials are given on the command line as system properties, then the properties `docker.pull.username` / `docker.pull.password` and `docker.push.username` / `docker.push.password` are

used for pull and push operations, respectively (when given). Either way, the standard lookup algorithm as described in the previous section is used as fallback.

12.2. OpenShift Authentication

When working with the default registry in OpenShift, the credentials to authenticate are the OpenShift username and access token. So, a typical interaction with the OpenShift registry from the outside is:

```
oc login
...
mvn -Ddocker.registry=docker-registry.domain.com:80/default/myimage \
    -Ddocker.username=$(oc whoami) \
    -Ddocker.password=$(oc whoami -t)
```

(note, that the image's user name part ("default" here") must correspond to an OpenShift project with the same name to which you currently connected account has access).

This can be simplified by using the system property `docker.useOpenShiftAuth` in which case the plugin does the lookup. The equivalent to the example above is

```
oc login
...
mvn -Ddocker.registry=docker-registry.domain.com:80/default/myimage \
    -Ddocker.useOpenShiftAuth
```

Alternatively the configuration option `<useOpenShiftAuth>` can be added to the `<authConfig>` section.

For dedicated *pull* and *push* configuration the system properties `docker.pull.useOpenShiftAuth` and `docker.push.useOpenShiftAuth` are available as well as the configuration option `<useOpenShiftAuth>` in an `<pull>` or `<push>` section within the `<authConfig>` configuration.

12.3. Password encryption

Regardless which mode you choose you can encrypt password as described in the Maven documentation. Assuming that you have setup a *master password* in `~/.m2/security-settings.xml` you can create easily encrypt passwords:

Example

```
$ mvn --encrypt-password
Password:
{QJ6wvuEfacMHklqsmrtrn1/ClOLqLm8hB7yUL23KOKo=}
```

This password then can be used in `authConfig`, `docker.password` and/or the `<server>` setting configuration. However, putting an encrypted password into `authConfig` in the `pom.xml` doesn't make

much sense, since this password is encrypted with an individual master password.

12.4. Extended Authentication

Some docker registries require additional steps to authenticate. Amazon ECR requires using an IAM access key to obtain temporary docker login credentials. The `docker:push` and `docker:pull` goals automatically execute this exchange for any registry of the form *<awsAccountId>* **.dkr.ecr.** *<awsRegion>* **.amazonaws.com**, unless the `skipExtendedAuth` configuration (`docker.skip.extendedAuth` property) is set true.

You can use any IAM access key with the necessary permissions in any of the locations mentioned above except `~/.docker/config.json`. Use the IAM **Access key ID** as the username and the **Secret access key** as the password.

Chapter 13. Migration from version 2

This version 3 of f8-m-p is using a completely new configuration syntax compared to version 2.

If you have a maven project with a 2.x fabric8-maven-plugin then we recommend you run the mvn fabric8:migrate goal directly on your project to do the migration:

```
# in a fabric8-maven-plugin 2.x project
mvn fabric8:migrate
# now the project is using 3.x or later
```

Once the project is migrated to 3.x or later of the fabric8-maven-plugin you can then run this `fabric8:setup` goal at any time to update to the latest plugin and goals.

Chapter 14. FAQ

14.1. General questions

14.1.1. How do I define an environment variable?

The easiest way is to add a `src/main/fabric8/deployment.yml` file to your project containing something like:

```
spec:
  template:
    spec:
      containers:
        -env:
          - name: FOO
            value: bar
```

The above will generate an environment variable `$FOO` of value `bar`

For a full list of the environments used in java base images, see this list

14.1.2. How do I define a system property?

The simplest way is to add system properties to the `JAVA_OPTIONS` environment variable.

For a full list of the environments used in java base images, see this list

e.g. add a `src/main/fabric8/deployment.yml` file to your project containing something like:

```
spec:
 template:
   spec:
     containers:
       - env:
         - name: JAVA_OPTIONS
           value: "-Dfoo=bar -Dxyz=abc"
```

The above will define the system properties `foo=bar` and `xyz=abc`

14.1.3. How do I mount a config file from a ConfigMap?

First you need to create your `ConfigMap` resource via a file `src/main/fabric8/configmap.yml`

```
data:
  application.properties: |
    # spring application properties file
    welcome = Hello from Kubernetes ConfigMap!!!
    dummy = some value
```

Then mount the entry in the `ConfigMap` into your `Deployment` via a file `src/main/fabric8/deployment.yml`

```
metadata:
  annotations:
    configmap.fabric8.io/update-on-change: ${project.artifactId}
spec:
  replicas: 1
  template:
    spec:
      volumes:
        - name: config
          configMap:
            name: ${project.artifactId}
            items:
            - key: application.properties
              path: application.properties
      containers:
        - volumeMounts:
            - name: config
              mountPath: /deployments/config
```

Here is an example quickstart doing this

Note that the annotation `configmap.fabric8.io/update-on-change` is optional; its used if your application is not capable of watching for changes in the `/deployments/config/application.properties` file. In this case if you are also running the configmapcontroller then this will cause a rolling upgrade of your application to use the new `ConfigMap` contents as you change it.

14.1.4. How do I use a Persistent Volume?

First you need to create your `PersistentVolumeClaim` resource via a file `src/main/fabric8/foo-pvc.yml` where `foo` is the name of the `PersistentVolumeClaim`. It might be your app requires multiple vpersistent volumes so you will need multiple `PersistentVolumeClaim` resources.

```
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

Then to mount the `PersistentVolumeClaim` into your `Deployment` create a file `src/main/fabric8/deployment.yml`

```
spec:
  template:
    spec:
      volumes:
      - name: foo
        persistentVolumeClaim:
          claimName: foo
      containers:
      - volumeMounts:
        - mountPath: /whatnot
          name: foo
```

Where the above defines the `PersistentVolumeClaim` called `foo` which is then mounted into the container at `/whatnot`

Here is an example application