# fabric8io/fabric8-maven-plugin

Roland Huß

Version 3.1.42, 2016-09-21

# fabric8-maven-plugin

# 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 builds and benefits from the build information 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 requirementes 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.

This introduction will explain how f8-m-p supports these tasks and demonstrates the different configuration models with examples.

## 1.1. Building Images

The **fabric8:build** goal is for creating Docker images which carry the actual application and which can be deployed on Kubernetes or OpenShift. It is easy to include build artifacts and their dependencies. The plugin uses the assembly descriptor format from the maven-assembly-plugin to specify the content which will be added to a sub-directory in the image (`/deployments` by default). Images that are built with this plugin 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 contacted directly or an OpenShift Docker Build is performed.

A special [fabric8:watch] goal allows for reacting on code changes and automatic recreation of images or copying new artifacts into running container.

These image related features are inherited from the fabric8io/docker-maven-plugin which is transparently included in this plugin.

## 1.2. Kubernetes and OpenShift Resources

With **fabric8:resource** Kubernetes and OpenShift resource descriptors can be created from the build information for creating the corresponding resource object. These files are packaged within the Maven artifacts created and can be deployed to a running orchestration platform with **fabric8:deploy**.

You only specify a fragment of the real resource descriptors which will be enriched by this plugin with various extra informations taken from the build. This drastically can reduce boilerplate code for common scenarios. It is also possible to auto-create resource objects like services or replica-set without explicitly declaring it.

## 1.3. Configuration

In order to capture many use case scenarios, there are three levels of configuration:

- **Zero-Config** mode makes some decisions based 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 plain configuration file with all their capabilities, but also to add project specific build information and avoid boilerplate code.

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 creatin 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. The 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 provide configuration can be still enhanced by Enhancers which is useful for adding e.g. labels and annotation containing build or other information. |
| **Resource Fragments** and **Dockerfiles** | Like the docker-maven-plugin f8-m-p supports external Dockerfiles too, which are referenced from the plugin configuration. | Resource descriptors can be provied 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 informations. |

## 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>3.1.42</version>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId> ①
    <version>1.3.6.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>3.1.42</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-alpine-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 apllication 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 Socker or with the URL set in `DOCKER_HOST`). Alternatively, when using `mvn -Dfabric8.mode=openshift package fabric8:build` and connected to an OpenShift cluster, 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`:

*Example for lifecycle bindings*

```
<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 implemented only partially and not recommended to use right now.

Altough the Zero-config mode with its generators can be tweaked with options up to a certain degree. In many cases more flexibility and power is required, though. For this an XML based plugin configuration can be use, much similar to the XML configuration used by `docker-maven-plugin`.

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

- A global configuration options are responsible for tuning the behaviour of plugin goals
- `<images>` section which defines the Docker images to build. It has the same syntax as the similar configuration of `docker-maven-plugin` (except that `<run>` and `<external>` sub-elements are ignored)
- `<resource>` is used to defined the resource descriptors for deploying on an OpenShift or Kuberneres cluster.
- `<generator>` is for configuring generators which are responsible for creating images. Generators are used as an alternative to a dedicates `<images>` section.
- `<enricher>` is used to configure 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 in the next example

*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
    <group>quickstarts</group>
  </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 Container 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. For creating OpenShift resource descriptor an automatic conversion will happen, e.g. from Kubernetes Deployment 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 is own file, which contains some skeleton of a resource description. The plugin will pick up the resource, enriches it and the combines all to a single `kubernetes.yml` and `openshift.yml`. Within these descriptor files you are can freely use any Kubernetes feature. Note, that in order to support simultaneously both OpenShift and Kubernetes, there is currently no way to specify OpenShift feature only this way (but this might change).

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/mainfabric8`.

*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 expected for each Kubernetes resource object. This section will be created automatically by fabric8-maven-plugin. The object's `Kind`, if not given, will be extracted from the filename. In this case its a `Deployment` because the file is called `deployment.xml`. For each supported resource type such a mapping exists. In addition you could specify a name in like in `myapp-deployment.xml` to give the resource a fixed name. Otherwise it will be automatically extracted from project information (i.e. the artificact id).

Here also the reference to the image is missing. In this case it will be automatically connected to the image you are building with this plugin (And you already know, that the image definition comes either from a generator or by a dedicated image plugin configuration).

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.

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. 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>3.1.42</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 3. Goals

This plugin supports the following goals, which are explained in the next sections:

*Table 2. Plugin Goals*

| Goal | Description |
|---|---|
| **fabric8:build** | Build images |
| **fabric8:push** | Push images to a registry |
| **fabric8:resource** | Create Kubernetes or OpenShift resource descriptors |
| **fabric8:helm** | Create a Helm Chart |
| **fabric8:deploy** | Deploy resources decriptors to a cluster |
| **[fabric8:watch]** | Watch for doing rebuilds and redeployments |

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

*Table 3. 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 |

## 3.1. fabric8:setup

This goal is for setting up maven projects to use f8-m-p. It can be used to set up afresh this plugin in the `pom.xml` or to update an existing configuration.

To use this goal type:

```
mvn io.fabric8:fabric8-maven-plugin:3.1.42:setup
```

Now your `pom.xml` is modified to use f8-m-p. The plugin will be bound some goals to Maven execution phases, so you can test it by running:

```
mvn clean install
```

After this, a Docker image from your application is created (if some of the default generators could detect your setup) and the Kubernetes and OpenShift descriptors are created in `target/classes/META-INF/fabric8`.

*Manual Setup*

Alternatively, if you prefer to do it all by hand, then add the following to your `pom.xml` file:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.1.42</version>

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

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

The setup can be influenced with the following configuration options:

*Table 4. Setup configuration*

| Element | Description | Property |
|---|---|---|
| **updateVersion** | If set to `true` then an already existing plugin configuration will be updated. Otherwise an existing configuration is left untouched. Default is `true`. | `updateVersion` |

| Element | Description | Property |
|---|---|---|
| **useVersionProperty** | Whether we should use a version property for the plugin which is defined in a dedicates `<properties>` section with the name `fabric8.maven.plugin.version` | `useVersionProperty` |
| **generateBackupPoms** | Controls whether a backup pom should be created when the `pom.xml` is modified. Ddefault is `true`. | `generateBackupPoms` |
| **backupPomFileName** | Name of the backup file to create. Default is `${basedir}/pom.xml-backup` | `backupPomFileName` |

## 3.2. fabric8:cluster-start

This goal will start a local kubernetes cluster for local development.

```
mvn fabric8:cluster-start
```

This will internally invoke the **fabric8:install** goal to ensure that all the required binaries are installed (like gofabric8  and for kubernetes: kubectl and minikube or for OpenShift: oc and minishift)

By default the binaries are installed in `~/.fabric8/bin`

### 3.2.1. Using OpenShift

By default `fabric8:cluster-start` will use `minikube` to create a local single node kubernetes cluster. To specify OpenShift use:

```
mvn fabric8:cluster-start -Dfabric8.cluster.kind=openshift
```

This will then use `minishift` instead to create a single node local OpenShift cluster.

### 3.2.2. Configure apps

By default the cluster contains only the fabric8 developer console as often developers laptops don't have lots of RAM.

If you want to deploy the full fabric8 platform (with Nexus, Jenkins, Gogs, JBoss Forge etc) then use the following command:

```
mvn fabric8:cluster-start -Dfabric8.cluster.app=platform
```

### 3.2.3. Configure cluster resources

You can specify the number of CPUs or memory via additional parameters:

```
mvn fabric8:cluster-start -Dfabric8.cluster.cpus=2 -Dfabric8.cluster.memory=4096
```

The above configures `2` CPUs and `4Gb` of memory

### 3.2.4. Stop

You can stop the cluster at any time via **fabric8:cluster-stop**

```
mvn fabric8:cluster-stop
```

Once stopped you can restart again with all the images, resources and pods intact later on by running `fabric8:cluster-start` again

```
mvn fabric8:cluster-start
```

## 3.3. fabric8:cluster-stop

This goal will stop a local kubernetes cluster.

This goal  stops the VM running the local cluster so it will free up resources on your machine (memory + CPU) though the VM is not destroyed; it can restarted.

```
mvn fabric8:cluster-stop
```

### 3.3.1. Restarting

You can restart the cluster at any time via **fabric8:cluster-start**

```
mvn fabric8:cluster-start
```

Once restarted all the images, resources and pods should come back

### 3.3.2. Deleting

If you wish to destroy the cluster VM and all the data inside it then you can pass the `fabric8.cluster.delete` parameter with a value of `true`:

```
mvn fabric8:cluster-stop -Dfabric8.cluster.delete=true
```

## 3.4. fabric8:run

This goal builds your application (generating the docker image and kubernetes manifest), deploys it on the current kubernetes cluster then tails the logs of the first pod that starts until you hit `Ctrl+C` then the application is stopped.

```
mvn fabric8:run
```

So this goal feels very much like the `run` goal in other maven plugins like spring-boot, tomcat, jetty, wildfly etc.

You can think of this goal as being similar to performing:

```
mvn fabric8:deploy
mvn fabric8:logs
...
^C
mvn fabric8:stop
```

If you wish to **fabric8:undeploy** on the `Ctrl+C` keypress you can pass in the `fabric8.onExit` goal:

```
mvn fabric8:run -D fabric8.onExit=undeploy
```

If you prefer Ctrl-C to just terminate the log tailing but leave your app running you can use:

```
mvn fabric8:run -D fabric8.onExit=
```

Though its maybe just simpler to do:

```
mvn fabric8:deploy fabric8:log
```

**Note** that you must have the **fabric8:resource** and **fabric8:build** goals bound to your executions in your pom.xml. e.g. like this:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.1.42</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>
```

## 3.5. fabric8:deploy

This is the main goal for building your docker image, generating the kubernetes resources and deploying them into the cluster.

```
mvn fabric8:deploy
```

Effectively this builds your project then invokes these goals:

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

**Note** that you must have the **fabric8:resource** and **fabric8:build** goals bound to your executions in your pom.xml. e.g. like this:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.1.42</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>
```

## 3.6. 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
```

## 3.7. fabric8:start

This goal starts the app that you deployed via the **fabric8:deploy** goal and then subsequently stopped via **fabric8:stop**.

```
mvn fabric8:start
```

This goal iterates through all scaleable resources generated by the **fabric8:resource** goal and scales them to a *replica count* of`1` so that by the time this goal completes there should be a single running pod of this app.

### 3.7.1. Scaling

You can also use the `fabric8:start` goal to scale the number of pods running your application via passing the `fabric8.replicas` parameter:

```
mvn fabric8:start  -Dfabric8.replicas=2
```

After the above you should see 2 pods running for your application.

From the command line you can use the `kubectl` binary to watch pods come and go via the following:

```
kubectl get pod -w
```

The `-w` flag watches the resource and keeps updating the console as things change. Leaving off the `-w` just lists the current pods only then `kubect` terminates.

## 3.8. fabric8:stop

This goal stops the app that you deployed via the **fabric8:deploy** goal.

```
mvn fabric8:stop
```

This goal iterates through all scaleable resources generated by the **fabric8:resource** goal and scales them to a *replica count* of `0` so that by the time this goal completes there are no running pods for this app.

## 3.9. 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
```

## 3.10. 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.

## 3.11. fabric8:import

This goal imports the current project into the [fabric8 console] so that you can browse the source code via the web interface and associate a [Continuous Delivery] pipeline with the project to start automatic Continuous Deployment.

If you have a project on the file system then type:

```
mvn fabric8:import
```

This goal assumes you have already enabled the `fabric8 maven plugin`. If you have not done so already then invoke the **fabric8:setup** goal first.

If no git repository exists yet for the current project then a new git repository will be created and the current code pushed into it so that it can then be built with Jenkins.

If the current project is a git clone from a remote repository then this goal will also try to setup a `Secret` in kubernetes to keep track of the user name and password/access code or in the case of SSH protocol with git the SSH key pairs.

## 3.12. fabric8:resource

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

### 3.12.1. 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
          <organisation>unesco</organisation> ②
        </all>
        <service> ③
          <persistent>true</persistent>
          <database>mysql</dabatabase>
        </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 lables 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 5. 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. |

## 3.13. 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>3.1.42</version>

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

# 3.14. fabric8:build

This goal is for building Docker images. Images can be build in two ways which depend on the `mode` (property: `fabric8.mode`). This mode can have be either `kubernetes` for a standard Docker build (the default) or `openshift` for an OpenShift build.

By default the mode is set to `auto`. In this case the plugin tries to detect which kind of build should be performed by contaction the API server. If this fails or if no cluster access is conigured e.g. with `oc login` then the mode is set to `kubernetes`.

## 3.14.1. 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 their is no need to distribute images.

## 3.14.2. OpenShift Build

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

There are two kind of binary builds supporte by this plugin, which can be selected with the configuration option `buildStrategy` (property `fabric8.build.strategy`)

*Table 6. 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 configuration option `buildRecreate` (property `fabric8.build.recreate`) is set to a value as described in Configuration. Also if the provided build strategy is different than for 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 referenced directly from a Deployment Configuration objects created by **fabric8:resource**.

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 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 a `<images>` section which defines how the one or more 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 mostly the Assembly is relevant because it depends on the buider image how to interpret the content of the uploaded `docker.tar`.

### 3.14.3. 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 flexibel 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 7. 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.buildStrategy` |
| **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 |
|---|---|---|
| **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 | `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` | `docker.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` |

### 3.14.4. 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 8. 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. |
| **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 9. 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

### 3.14.5. 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*

Alternatively an external Dockerfile template can be used. This mode is switch on by using one of these two configuration options within the `<build>` configuration section.

- **dockerFileDir** specifies a directory containing a `Dockerfile` that will be used to create the image.
- **dockerFile** specifies a specific Dockerfile. The `dockerFileDir` is set to the directory containing the file.

If `dockerFileDir` is a relative path looked up in `${project.basedir}/src/main/docker`. You can make easily an absolute path by prefixing with `${project.basedir}`.

Any additional files located in the `dockerFileDir` directory will also be added to the build context as well as any files specified by an assembly. However, you still need to insert `ADD` or `COPY` directives yourself into the Dockerfile.

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.

> For the future it is planned to introduce special keywords lile `DMP_ADD_ASSEMBLY` which can be used in the Dockerfile template to placing the configuration resulting from the additional configuration.

The following example uses a Dockerfile in the directory `src/main/docker/demo`:

*Example*

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

All build relevant configuration is contained in the `<build>` section of an image configuration. In addition to `<dockerFileDir>` and `<dockerFile>` the following configuration options are available:

*Table 10. Build configuration*

| 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 |
| **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. |
| **nocache** | Don't use Docker's build cache. This can be overwritten by setting a system property `docker.nocache` when running Maven. |
| **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. |
| **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. |
| **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`. |
| **labels** | Labels as described in Setting Environment Variables and Labels. |

| Element | Description |
|---|---|
| **maintainer** | The author (`MAINTAINER`) field for the generated image |
| **ports** | The exposed ports which is a list of `<port>` elements, one for each port to expose. |
| **runCmds** | Commands to be run during the build process. It contains **run** elements which are passed to the shell. 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. |
| **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. |
| **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`. |
| **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. |
| **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. |
| **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>

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

  <assembly>
    <mode>dir</mode>
    <basedir>/opt/demo</basedir>
    <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

### 3.14.6. 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 11. Assembly Configuration*

| Element | Description |
|---|---|
| **basedir** | Directory under which the files and artifacts contained in the assembly will be copied within the container. The default value for this is `/maven`. |
| **inline** | Inlined assembly descriptor as described in Assembly Descriptor below. |
| **descriptor** | Path to an assembly descriptor file, whose format is described Assembly Descriptor below. |

| Element | Description |
| --- | --- |
| **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. |
| **exportBasedir** | Specification whether the `basedir` should be exported as a volume. This value is `true` by default except in the case the `basedir` 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. |
| **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 12. 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 `baseDir` (which is `/maven` by default).

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

If the assembly references the artifact to build with this pom, it is required that the `package` phase is included in the run. This happens either automatically when the `fabric8:build` target is called as part of a binding (e.g. is `fabric8:build` is bound to the `pre-integration-test` phase) or it must be ensured when called on the command line:

*Example*

```
mvn package fabric8:build
```

This is a general restriction of the Maven lifecycle which applies also for the `maven-assembly-plugin` itself.

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.

*Example*

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

### 3.14.7. 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>
```

### 3.14.8. 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.

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

*Table 13. 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>
     <args>java</args>
     <args>-jar</args>
     <args>/opt/demo/server.jar</args>
   </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

### 3.14.9. 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.

## 3.15. 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 `image` (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.

*Table 14. 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` |

## 3.16. fabric8:install

Ensures that the fabric8 binaries are installed on the current machine such as gofabric8 and for kubernetes: kubectl and minikube or for OpenShift: oc and minishift

```
mvn fabric8:install
```

An alternative is to just run the **fabric8:cluster-start** goal to install the binaries and start a local cluster

By default the binaries are installed in `~/.fabric8/bin`

## 3.17. fabric8:helm

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 15. 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` |
| **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 | |

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).

## 3.18. fabric8:helm-index

Generates a Manifest index file by querying a maven repository to find all the Kubernetes and OpenShift manifests available and their releases.

## 3.19. fabric8:app-catalog

Generates an App Catalog for kubernetes and openshift

On OpenShift this just means getting the `openshift.yml` and if its not a `Template` wrapping it in an empty `Template`.

For Kubernetes this means checking if there is a kubernetes `template.yaml` and if so wrapping that in a `ConfigMap` otherwise it uses the regular `kubernetes.yaml` file.

## 3.20. fabric8:distro

Generates a tarball of all the dependent kubernetes and openshift templates

# Chapter 4. Generators

The usual way to define Docker images is with the plugin configuration as explained in **fabric8:build**. This can either be done completly within the pom.xml or by refering to an external Dockerfile. Since fabric8-maven-plugin includes our docker-maven-plugin the way how 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 type 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 an 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 however will focus on existing generators and how you can configure them.

By default, this plugin includes already a set of generators, which are explained in detail below. These generators are enabled by default, but you can easil 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.

Generators can be configured with a `<generator>` section. The following sub-elements are supported.

*Table 16. 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. |
| `<excludes>` | Holds one or more `<exclude>` elements with generator names to exclude. This mean 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. |

Beside the specifying generator configuration in the plugin's configuration it can be set 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.

fabric8-maven-plugin comes with a set of default generators which are described in the next secion Fabric8 Generators. These are enabled by default. In addition, custom generators can be easily added by simply declaring a compile scoped dependency in the `pom.xml`. This and how a generator can be implemented is described in Generator API.

# 4.1. Default Generators

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

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

*Table 17. Common generator options*

| Element | Description | Property |
|---|---|---|
| **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 |
|---|---|---|
| **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` |
| **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` |
| **merge** | 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 by a generator which has been run previously. | |

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

### 4.1.1. Spring Boot

The name of this generator is `spring-boot` and gets activated when it finds a `spring-boot-maven-plugin` among the configured plugins. This plugin can be also a included as a dependency. It will use the following base image by default, but as explained above and can be changeda `from` configuration.

*Table 18. Spring-Boot Base Images*

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

These images refer always 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).

The following additional configuration options can be set:

*Table 19. Spring-Boot configuration options*

| Element | Description | Default |
|---|---|---|
| **webPort** | Port to expose as service | 8080 |
| **jolokiaPort** | Port of the Jolokia agent exposed by the base image | 8778 |
| **prometheusPort** | Port of the Prometheus jmx_exporter exposed by the base image | 9779 |

### 4.1.2. Java Applications

This generator is named `java-exec` and is responsible to start up arbitrary Java application. It kicks

in if eithe the main class is explicitely configured in this generator's configuration or when it finds a singel class with a `public static void main(String[] args)` method. If it finds more than one class, the generator is a no op.

It use the same default images as the Spring Boot generator.

Beside the common configuration parameters described in the table Commong Generator Options it knows the following additional configuration options:

*Table 20. Java Application configuration options*

| Element | Description | Default |
|---|---|---|
| **mainClass** | Main class to call. If not given 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, this generator does nothing. | |
| **webPort** | Port to expose as service | 8080 |
| **jolokiaPort** | Port of the Jolokia agent exposed by the base image | 8778 |
| **prometheusPort** | Port of the Prometheus jmx_exporter exposed by the base image | 9779 |

### 4.1.3. Karaf

## 4.2. 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 5. 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 similarities to Generatos:

- Each Enricher has a unique name.
- Enrichers are looked up automatically from dependencies provied 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 21. 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. |
| `<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. |

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.

## 5.1. 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.

*Table 22. 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. |
| 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-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-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. |

### 5.1.1. Standard Enrichers

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

**fmp-controller**

**fmp-service**

**fmp-image**

**fmp-name**

**fmp-project**

**fmp-git**

**fmp-dependency**

### 5.1.2. 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**

## 5.2. Enricher API

*howto write your own enricher and install them*

# Chapter 6. Profiles

Profiles can be used to combine a set of enrichers and generators and gives this combination a referenceable name. 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 added in these `profiles.yml` descriptors.

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

*Profile Definition*

```
- name: my-spring-boot-apps
  generator:
    includes:
      - spring-boot
  enrichers:
    includes:
      # Default Deployment object
      - fmp-controller
      # Add a default service
      - fmp-service
  config:
    fmp-service:
      # Expose service as NodePort
      type: NodePort
```

This profile then can be referenced from various plugin goals. The easiest way is to add the profile as top-level configuration parameter.

*Specification of profile in plugin configuration*

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

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.

# Chapter 7. Access configuration

## 7.1. 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.

## 7.2. 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 8. 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 `docker.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 9. 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 three different ways for providing credentials:

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

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 accorunt '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.

## 9.1. 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.

## 9.2. OpenShift Authentication

When working with the default registry in OpenShift, the credentials to authtenticate 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 dedicted *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.

## 9.3. 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 encrypted 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.

# Chapter 10. Migration from version 2

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