fabric8io/fabric8-maven-plugin

Roland Huß

Version 3.0-SNAPSHOT, 2016-07-26

fabric8-maven-plugin

Chapter 1. Introduction

The **fabric8-maven-plugin** brings your Java applications to Kubernetes and OpenShift. It provides a tight integration into Maven builds and benefits from the build information already given. The main tasks where fabric8-maven-plugin can help is in building images and creating Kubernetes and OpenShift resource descriptors.

1.1. Building Images

The **fabric8:build** goal is for creating Docker images which carrythe 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 Resource Descriptors

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 festures is provided.

- **Kubernetes & OpenShift resource descruptors** are user provided YAML fragments 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.

1.4. Examples

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

1.4.1. Zero-Config

This minimal example `pom.xml` shows how a simple spring boot application can be dockerized and prepared for Kubernetes and OpenShift:

Example

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

  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-sample-zero-config</artifactId>
  <version>3.0-SNAPSHOT</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>3.0-SNAPSHOT</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 repackagign 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 sophistacted 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

1.4.3. Enhanced YAML Descriptors

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>3.0-SNAPSHOT</version>

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

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

When working with this plugin you can use an own packaging with a specialized lifecycle in order to keep your pom files small. Three packaging variants are available:

The rest of this manual is now about how to configure the plugin for your images.

Chapter 3. Goals

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

Table 1. Plugin Goals

Goal	Description
fabric8:build	Build images
fabric8:push	Push images to a registry
fabric8:resource	Create Kubernetes or OpenShift resource descriptors
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 2. Strategies

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 a `BuildConfig` * Starts an OpenShift build with a `docker.tar` as input * Creates an `ImageStream` which can be referenced by the deployment descriptors
Deploy	`fabric8:deploy` * Applies a Kubernetes resource descriptor to cluster	`fabric8:deploy` * Applies an OpenShift resource descriptor to a cluster

3.1. fabric8:resource

3.2. fabric8:build

This goal will build all images which have a `<build>` configuration section, or, if the global configuration variable `image` (property: `docker.image`) is set, only the images contained in this variable (comma separated) will be built.

Images can be build in two different ways:

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>
```

3.2.1. Configuration

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 3. Build configuration

Element	Description
args	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
assembly	specifies the assembly configuration as described in Build Assembly
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.
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.
ulimits	ulimits for the container. This list contains `<ulimit>` elements which three sub elements: * `<name>` : The ulimit to set (e.g. `memlock`). Please refer to the Docker documentation for the possible values to set * `<hard>` : The hard limit * `<soft>` : The soft limit See below for an example.
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>
  <ulimits>
    <ulimit>
       <name>memlock</name>
       <hard>-1</hard>
       <soft>-1</soft>
    </ulimit>
  <ulimits>
  <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

3.2.2. 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 4. 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.
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 5. 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`.

3.2.3. 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>
```

3.2.4. 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 6. 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>
```

3.2.5. 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.

3.3. fabric8:push

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 7. 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. 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`

3.4. fabric8:deploy

3.5. fabric8:watch

When developing and testing applications you will often have to rebuild Docker images and restart containers. Typing `fabric8:build` and `fabric8:start` all the time is cumbersome. With `fabric8:watch` you can enable automatic rebuilding of images and restarting of containers in case of updates.

`fabric8:watch` is the top-level goal which perform these tasks. There are two watch modes, which can be specified in multiple ways:

- `build` : Automatically rebuild one or more Docker images when one of the files selected by an assembly changes. This works for all files included directly in `assembly.xml` but also for arbitrary dependencies.

Example

```
$ mvn package fabric8:build fabric8:watch -Ddocker.watchMode=build
```

This mode works only when there is a `<build>` section in an image configuration. Otherwise no automatically build will be triggered for an image with only a `<run>` section. Note that you need the `package` phase to be executed before otherwise any artifact created by this build can not be included into the assembly. As described in the section about `fabric8:start` this is a Maven limitation. * `run` : Automatically restart container when their associated images changes. This is useful if you pull a new version of an image externally or especially in combination with the `build` mode to restart containers when their image has been automatically rebuilt. This mode works reliably only when used together with `fabric8:start`.

Example

```
$ mvn fabric8:start fabric8:watch -Ddocker.watchMode=run
```

- `both` : Enables both `build` and `run`. This is the default.
- `none` : Image is completely ignored for watching.
- `copy` : Copy changed files into the running container. This is the fast way to update a container, however the target container must support hot deply, too so that it makes sense. Most application servers like Tomcat supports this.

The mode can also be `both` or `none` to select both or none of these variants, respectively. The default is `both`.

`fabric8:watch` will run forever until it is interrupted with `CTRL-C` after which it will stop all containers. Depending on the configuration parameters `keepContainer` and `removeVolumes` the stopped containers with their volumes will be removed, too.

When an image is removed while watching it, error messages will be printed out periodically. So don't do that ;-)

Dynamically assigned ports stay stable in that they won't change after a container has been stopped and a new container is created and started. The new container will try to allocate the same ports as the previous container.

If containers are linked together network or volume wise, and you update a container which other containers dependent on, the dependant containers are not restarted for now. E.g. when you have a "service" container accessing a "db" container and the "db" container is updated, then you "service" container will fail until it is restarted, too.

> A future version of this plugin will take care of restarting these containers, too (in the right order), but for now you would have to do this manually.

This maven goal can be configured with the following top-level parameters:

Table 8. Watch configuration

Element	Description	Property
watchMode	Watch mode specifies what should be watched * `build` : Watch changes in the assembly and rebuild the image in case * `run` : Watch a container's image whether it changes and restart the container in case * `copy` : Changed files are copied into the container. The container can be either running or might be already exited (when used as a *data container* linked into a *platform container*). Requires Docker >= 1.8. * `both` : `build` and `run` combined * `none` : Neither watching for builds nor images. This is useful if you use prefactored images which won't be changed and hence don't need any watching. `none` is best used on an per image level, see below how this can be specified.	`docker.watchMode`
watchInterval	Interval in milliseconds how often to check for changes, which must be larger than 100ms. The default is 5 seconds.	`docker.watchInterval`
watchPostGoal	A maven goal which should be called if a rebuild or a restart has been performed. This goal must have the format `<pluginGroupId>:<pluginArtifactId>:<goal>` and the plugin must be configured in the `pom.xml`. For example a post-goal `io.fabric8:fabric8:delete-pods` will trigger the deletion of PODs in Kubernetes which in turn triggers are new start of a POD within the Kubernetes cluster. The value specified here is the the default post goal which can be overridden by `<postGoal>` in a `<watch>` configuration.	
watchPostExec	A command which is executed within the container after files are copied into this container when `watchMode` is `copy`. Note that this container must be running.	
keepRunning	If set to `true` all container will be kept running after `fabric8:watch` has been stopped. By default this is set to `false`.	`docker.keepRunning`
keepContainer	As for `fabric8:stop`, if this is set to `true` (and `keepRunning` is disabled) then all container will be removed after they have been stopped. The default is `true`.	`docker.keepContainer`
removeVolumes	if set to `true` will remove any volumes associated to the container as well. This option will be ignored if either `keepContainer` or `keepRunning` are `true`.	`docker.removeVolumes`

Image specific watch configuration goes into an extra image-level `<watch>` section (i.e. `<image><watch>...</watch></image>`). The following parameters are recognized:

Table 9. Watch configuration for a single image

Element	Description
mode	Each image can be configured for having individual watch mode. These take precedence of the global watch mode. The mode specified in this configuration takes precedence over the globally specified mode.
interval	Watch interval can be specified in milliseconds on image level. If given this will override the global watch interval.

Element	Description
postGoal	Post Maven plugin goal after a rebuild or restart. The value here must have the format `<pluginGroupId>:<pluginArtifactId>:<goal>` (e.g. `io.fabric8:fabric8:delete-pods`)
postExec	Command to execute after files are copied into a running container when `mode` is `copy`.

Here is an example how the watch mode can be tuned:

Example

```
<configuration>
   <!-- Check every 10 seconds by default -->
   <watchInterval>10000</watchInterval>
   <!-- Watch for doing rebuilds and restarts -->
   <watchMode>both</watch>
   <images>
      <image>
         <!-- Service checks every 5 seconds -->
         <alias>service</alias>
         ....
         <watch>
            <interval>5000</interval>
         </watch>
      </image>
      <image>
         <!-- Database needs no watching -->
         <alias>db<alias>
         ....
         <watch>
            <mode>none</mode>
         </watch>
      </image>
      ....
   </images>
</configuration>
```

Given this configuration

Example

```
mvn package fabric8:build fabric8:start fabric8:watch
```

You can build the service image, start up all containers and go into a watch loop. Again, you need the `package` phase in order that the assembly can find the artifact build by this project. This is a Maven limitation. The `db` image will never be watch since it assumed to not change while watching.

Chapter 4. Extensions

This plugin provides two major extensions hook how the creation of images and resources descriptors can be customized:

- **Generators** are used to auto create or customize image configuration when creating Docker images. They are a bit like Spring Boot Generator POMs as they can be enabled or disabled by declaring a Maven dependency. Generators are able to examine the build and to *detect* certain feature like whether Spring boot application is build or a plain war file. Depending on the collected informations a base image or the exposed ports are selected automatically for creating a image build configuration.
- **Enrichers** are a similar concept but for creating the Kubernetes resource descriptors. Enricher can add build meta data as labels, automatically create `ReplicaSet` or `Service` based on the image performed. Again, enrichers can be selectively switched on and off via declaring Maven dependencies or via the XML configuration. fabric8-maven-plugin already comes with a rich set of enrichers. Whereas **Generators** are only useful in the *Zero-Config* case, **Enrichers** make sense for any configuration variant.

The following sections described which Generators and Enrichers are available and how own customizations can be hooked in.

Unresolved directive in inc/_extensions.adoc - include::extensions/_generator.adoc[]

4.1. Enricher