fabric8io/docker-maven-plugin

Roland Huß

Version 0.15-SNAPSHOT, 2016-07-29

docker-maven-plugin

Chapter 1. Introduction

This is a Maven plugin for managing Docker images and containers. It focuses on two major aspects for a Docker build integration:

1.1. Building Images

One purpose of this plugin is to create Docker images holding the actual application. This is done with the **docker:build** goal. It is easy to include build artifacts and their dependencies into an image. Therefore, this plugin uses the assembly descriptor format from the maven-assembly-plugin to specify the content which will be added from a sub-directory in the image (`/maven` by default). Images that are built with this plugin can be pushed to public or private Docker registries with **docker:push**.

1.2. Running Containers

With this plugin it is possible to run completely isolated integration tests so you don't need to take care of shared resources. Ports can be mapped dynamically and made available as Maven properties to your integration test code.

Multiple containers can be managed at once, which can be linked together or share data via volumes. Containers are created and started with the **docker:start** goal and stopped and destroyed with the **docker:stop** goal. For integration tests both goals are typically bound to the the pre-integration-test and post-integration-test phase, respectively. It is recommended to use the maven-failsafe-plugin for integration testing in order to stop the docker container even when the tests fail.

For proper isolation, container exposed ports can be dynamically and flexibly mapped to local host ports. It is easy to specify a Maven property which will be filled in with a dynamically assigned port after a container has been started. This can then be used as parameter for integration tests to connect to the application.

1.3. Configuration

The plugin configuration contains a global part and a list of image-specific configuration within a `<images>` list, where each image is defined within a `<image>` tag. See below for an example.

The global part contains configuration applicable to all images like the Docker URL or the path to the SSL certificates for communication with the Docker Host.

Then, each specific image configuration has three parts:

- A general image part containing the image name and alias.
- A <build> configuration specifying how images are built
- A <run> configuration describing how containers should be created and started.

The `<build>` and `<run>` parts are optional and can be omitted.

1.4. Example

In the following examples two images are specified. One is the official PostgreSQL 9 image from Docker Hub, which internally is referenced with an alias "database". It only has a `<run>` section which declares that the startup should wait until the given text pattern is matched in the log output. Next is a "service" image, which has a `<build>` section. It creates an image which has artifacts and dependencies in the `/maven` directory (and which are specified with an assembly descriptor). Additionally it specifies the startup command for the container, which in this example fires up a microservice from a jar file copied over via the assembly descriptor. It also exposes port 8080. In the `<run>` section this port is mapped to a dynamically chosen port, and then assigned to the Maven property `${tomcat.port}`. This property could be used, for example, by an integration test to access this microservice. An important part is the `<links>` section which indicates that the image with the alias of "database" is linked into the "service" container, which can access the internal ports in the usual Docker way (via environment variables prefixed with `DB_`).

Images can be specified in any order and the plugin will take care of the proper startup order (and will bail out if it detects circular dependencies).

Example plugin configuration

```
<configuration>
  <images>
    <image>
      <alias>service</alias> ①
      <name>fabric8/docker-demo:${project.version}</name>

      <build> ②
         <from>java:8</from> ③
         <assembly>
           <descriptor>docker-assembly.xml</descriptor> ④
         </assembly>
         <cmd> ⑤
            <shell>java -jar /maven/service.jar</shell>
         </cmd>
      </build>

      <run> ⑥
         <ports> ⑦
           <port>tomcat.port:8080</port>
         </ports>
         <wait> ⑧
           <http>
              <url>http://localhost:${tomcat.port}/access</url>
           </http>
           <time>10000</time>
         </wait>
         <links> ⑨
           <link>database:db</link>
         </links>
       </run>
    </image>

    <image>
      <alias>database</alias> ⑩
      <name>postgres:9</name>
      <run>
        <wait> ⑪
          <log>database system is ready to accept connections</log>
          <time>20000</time>
        </wait>
      </run>
    </image>
  </images>
</configuration>
```

① Image configuration for a Java service with alias "service" and name `fabric8/docker-demo:${project.version}`

② build configuration defines how a Docker image should be created

③ Base image, in this case `java:8`

④ Content of the image can be specified with an assembly descriptor

⑤ Default command to run when a container is created.

⑥ Run configuration defines how a container should be created from this image

⑦ Port mapping defines how container ports should be mapped to host ports

⑧ Wait section which is a readiness check when starting the service

⑨ Network link describes how this service's container is linked to the database container

⑩ Second image is a plain database image which is only needed for running (hence there is no `<build>` section). The alias is used in the network link section above

⑪ Wait until the corresponding output appears on stdout when starting the Docker container.

1.5. Features

Some other highlights, in random order:

- Auto pulling of images with a progress indicator
- Waiting for a container to startup based on time, the reachability of an URL, or a pattern in the log output
- Support for SSL Authentication and OpenShift credentials
- Docker machine support
- Flexible registry handling (i.e. registries can be specified as meta data)
- Specification of encrypted registry passwords for push and pull in ~/.m2/settings.xml (i.e., outside the pom.xml)
- Color output
- Watching on project changes and automatic recreation of image
- Properties as alternative to the XML configuration
- Support for Docker daemons accepting http or https request via TCP and for Unix sockets

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.

Example

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.15-SNAPSHOT</version>

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

  <!-- Connect start/stop to pre- and
       post-integration-test phase, respectively if you want to start
       your docker containers during integration tests -->
  <executions>
    <execution>
      <id>start</id>
      <phase>pre-integration-test</phase>
      <goals>
        <!-- "build" should be used to create the images with the
             artifact -->
        <goal>build</goal>
        <goal>start</goal>
      </goals>
    </execution>
    <execution>
      <id>stop</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>stop</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:

- **docker** : This binds `docker:build` to the package phase and `docker:start` / `docker:stop` to the pre- and post-integration phase respectively. Also `docker:push` is bound to the deploy phase.
- **docker-build** : Much like the *docker* packaging, except that there are no integration tests configured by default.
- **docker-tar** : Create a so called *Docker tar* archive which is used as the artifact and which later can be use for building an image. It contains essentially a `Dockerfile` with supporting files. See docker:source for more details.

These packaging definitions include the *jar* lifecycle methods so they are well suited simple Microservice style projects.

Example

```
<pom>
  <artifactId>demo</artifactId>
  <version>0.0.1</version>
  <packaging>docker</packaging>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>io.fabric8</groupId>
        <artifactId>docker-maven-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <images>
            <image>
            ...
            </image>
          </images>
        </configuration>
      </plugin>
    </plugins>
    ....
  <build>
</pom>
```

This will create the jar (if any), build the Docker images, start the configured Docker containers, runs the integration tests, stops the configured Docker container when you enter `mvn install`. With `mvn deploy` you can additionally push the images to a Docker configuration. Please note the `<extensions>true</extensions>` which is mandatory when you use a custom lifecycle.

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

Chapter 3. Global configuration

Global configuration parameters specify overall behavior like the connection to the Docker host. The corresponding system properties which can be used to set it from the outside are given in parentheses.

The docker-maven-plugin uses the Docker remote API so the URL of your Docker Daemon must somehow be specified. The URL can be specified by the dockerHost or machine configuration, or by the `DOCKER_HOST` environment variable.

Since 1.3.0, 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.

Table 1. 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.	
autoCreateCustomNetworks	Create automatically Docker networks during `docker:start` and remove it during `docker:stop` if you provide a custom network in the run configuration of an image. The default is `false`.	`docker.autoCreateCustomNetworks`
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`
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`
logDate	Date format which is used for printing out container logs. This configuration can be overwritten by individual run configurations and described below. The format is described in Logging.	`docker.logDate`
logStdout	For all container logging to standard output if set to `true`, regardless whether a `file` for log output is specified. See also Logging	`docker.logStdout`
machine	Docker machine configuration. See Docker Machine for possible values	
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`
outputDirectory	Default output directory to be used by this plugin. The default value is `target/docker` and is only used for the goal `docker: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.	
registry	Specify globally a registry to use for pulling and pushing images. See Registry handling for details.	`docker.registry`
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 `docker:build`	`docker.skip.build`

Element	Description	Property
skipPush	If set dont push any images even when `docker:push` is called.	`docker.skip.push`
skipRun	If set dont create and start any containers with `docker:start` or `docker:run`	`docker.skip.run`
skipTag	If set to `true` this plugin won't add any tags to images that have been built with `docker: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 `docker: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`

Example

```
<configuration>
   <dockerHost>https://localhost:2376</dockerHost>
   <certPath>src/main/dockerCerts</certPath>
   <useColor>true</useColor>
   .....
</configuration>
```

Docker Machine

This plugin supports also Docker machine (which must be installed locally, of course). A Docker machine configuration can be provided with a top-level `<machine>` configuration section. This configuration section knows the following options:

Table 2. Docker Machine Options

Element	Description
name	Docker machine's name. Default is `default`
autoCreate	if set to `true` then a Docker machine will automatically created. Default is `false`.
createOptions	Map with options for Docker machine when auto-creating a machine. See the docker machine documentation for possible options.

When no Docker host is configured or available as environment variable, then the configured Docker machine is used. If the machine exists but is not running, it is started automatically. If it does not exists but `autoCreate` is true, then the machine is created and started. Otherwise an error is printed. Please note, that a machine which has been created because of `autoCreate` gets never deleted by docker-maven-plugin. This needs to be done manually if required.

In absent of a `<machine>` configuration section the Maven property `docker.machine.name` can be used

to provide the name of a Docker machine. Similarly the property `docker.machine.autoCreate` can be set to true for creating a Docker machine, too.

You can use the property `docker.skip.machine` if you want to override the internal detection mechanism to always disable docker machine support.

Example

```
<!-- Work with a docker-machine -->
<configuration>
  <machine>
    <name>maven</name>
    <autoCreate>true</autoCreate>
    <createOptions>
      <driver>virtualbox</driver>
      <virtualbox-cpu-count>2</virtualbox-cpu-count>
    </createOptions>
  </machine>
   .....
</configuration>
```

Chapter 4. Image configuration

The plugin's configuration is centered around *images*. 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 3. 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 **docker: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.
run	Element which describe how containers should be created and run when **docker:start** is called. If this image is only used a *data container* (i.e. is supposed only to be mounted as a volume) for exporting artifacts via volumes this section can be missing.
external	Specification of external configuration as an alternative to this XML based configuration with `<run>` and `<build>`. It contains a `<type>` element specifying the handler for getting the configuration. See External configuration for details.

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 4. 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`)

Placeholder	Description
%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.

Either a `<build>` or `<run>` section must be present. These are explained in details in the corresponding goal sections.

Example

```
<configuration>
  ....
  <images>
    <image>
      <name>%g/docker-demo:0.1</name>
      <alias>service</alias>
      <run>....</run>
      <build>....</build>
    </image>
  </images>
</configuration>
```

Chapter 5. Maven Goals

This plugin supports the following goals which are explained in detail in the next sections.

Table 5. Plugin Goals

Goal	Description
docker:build	Build images
docker:start or **docker:run**	Create and start containers
docker:stop	Stop and destroy containers
docker:push	Push images to a registry
docker:watch	Watch for doing rebuilds and restarts
docker:remove	Remove images from local docker host
docker:logs	Show container logs
docker:source	Attach docker build archive to Maven project

Note that all goals are orthogonal to each other. For example in order to start a container for your application you typically have to build its image before. `docker:start` does **not** imply building the image so you should use it then in combination with `docker:build`.

All goals fork the lifecycle phase to call the lifecycle up to the **initialize** phase. This means you can bind other plugins to the phases **initialize** or **validate** for customization. By default no action is bound to these phases. See Introduction to the Build Lifecycle for more details about phases and lifecycles.

> Note however that when you bind the goals about to lifecycle phases this will result in the **initialize** and **validate** phase called twice. Your bound plugins should be able to handle this.

5.1. docker: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>
```

5.1.1. 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 6. 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 (`docker:build`) and how docker build archives are attached to this build as sources (`docker: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

5.1.2. 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 7. 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 8. 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 `docker:build` target is called as part of a binding (e.g. is `docker:build` is bound to the `pre-integration-test` phase) or it must be ensured when called on the command line:

Example

```
mvn package docker: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`.

5.1.3. 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 9. 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>
```

5.1.4. Build Args

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

- Set a system property when running Maven, eg.: `-Ddocker.buildArg.http_proxy=http://proxy:8001`. This is especially useful when using predefined Docker arguments for setting proxies transparently.
- Set a project property within the `pom.xml`, eg.:

Example

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

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

5.2. docker:start

This goal creates and starts docker containers. This goals evaluates the configuration's `<run>` section of all given (and enabled images). In order to switch on globally the logs **showLogs** can be used as global configuration (i.e. outside of `<images>`). If set it will print out all standard output and standard error messages for all containers started. As value the images for which logs should be shown can be given as a comma separated list. This is probably most useful when used from the command line as system property `docker.showLogs`.

Also you can specify `docker.follow` as system property so that the `docker:start` will never return but block until CTRL-C is pressed. That similar to the option `-i` for `docker run`. This will automatically switch on `showLogs` so that you can see what is happening within the container. Also, after stopping with CTRL-C, the container is stopped (but not removed so that you can make postmortem analysis).

By default container specific properties are exposed as Maven properties. These properties have the format `docker.container.<alias>.<prop>` where `<alias>` is the name of the container (see below) and `<prop>` is one of the following container properties:

Table 10. Properties provided

Property	Description
ip	Internal IP address of the container.
id	Container id

For example the Maven property `docker.container.tomcat.ip` would hold the Docker internal IP for a container with an alias "tomcat". You can set the global configuration **exposeContainerInfo** to an empty string to not expose container information that way or to a string for an other prefix than `docker.container`.

5.2.1. Configuration

The `<run>` configuration element knows the following sub elements:

Table 11. Run configuration

Element	Description
capAdd	List of `add` elements to specify kernel parameters to add to the container.
capDrop	List of `drop` elements to specify kernel parameters to remove from the container.
cmd	Command which should be executed at the end of the container's startup. If not given, the image's default command is used. See Startup Arguments for details.
domainname	Domain name for the container
dns	List of `host` elements specifying dns servers for the container to use
dnsSearch	List of `host` elements specifying dns search domains
entrypoint	Entry point for the container. See <<misc-startup, Startup Arguments] for details.
env	Environment variables as subelements which are set during startup of the container. They are specified in the typical maven property format as described Environment and Labels.
envPropertyFile	Path to a property file holding environment variables. If given, the variables specified in this property file overrides the environment variables specified in the configuration.
extraHosts	List of `host` elements in the form `host:ip` to add to the container's `/etc/hosts` file. Additionally, you may specify a `host` element in the form `host:host` to have the right side host ip address resolved at container startup.
hostname	Hostname of the container
labels	Labels which should be attached to the container. They are specified in the typical maven property format as described in Environment and Labels.
links	Network links for connecting containers together as described in Network Links.
log	Log configuration for whether and how log messages from the running containers should be printed. This also can configure the log driver to use. See Logging for a detailed description.
memory	Memory limit in bytes

Element	Description
memorySwap	Total memory usage (memory + swap); use -1 to disable swap.
namingStrategy	Naming strategy for how the container name is created: * **none** : uses randomly assigned names from docker (default) * **alias** : uses the `alias` specified in the `image` configuration. An error is thrown, if a container already exists with this name.
net	Network mode for the container: * **bridge** : Bridged mode with the default Docker bridge (default) * **host** : Share the Docker host network interfaces * **container:<alias or name>** : Connect to the network of the specified container * **<custom network>** : Use the specified custom network which must be created before with `docker network create`. Available for Docker 1.9 and newer. For more about the networking options please refer to the Docker documentation.
portPropertyFile	File path into which the mapped port properties are written. The format of this file and its purpose are also described in Port mapping
ports	Port mappings for exposing container ports to host ports.
privileged	If `true` give container full access to host
restartPolicy	Restart Policy
shmSize	Size of `/dev/shm` in bytes.
skip	If `true` disable creating and starting of the container. This option is best used together with a Maven property which can be set from the outside.
user	User used inside the container
volumes	Volume configuration for binding to host directories and from other containers. See Volumes for details.
wait	Condition which must be fulfilled for the startup to complete. See Wait for all possible ways to wait for a startup condition.
workingDir	Working directory for commands to run in

Example

```
<run>
  <env>
    <CATALINA_OPTS>-Xmx32m</CATALINA_OPTS>
    <JOLOKIA_OFF/>
  </env>
  <labels>
    <environment>development</environment>
    <version>${project.version}</version>
  </labels>
  <ports>
    <port>jolokia.port:8080</port>
  </ports>
  <links>
    <link>db</db>
  </links>
  <wait>
    <http>
      <url>http://localhost:${jolokia.port}/jolokia</url>
    </http>
    <time>10000</time>
  </wait>
  <log>
    <prefix>DEMO</prefix>
    <date>ISO8601</date>
    <color>blue</color>
  </log>
  <cmd>java -jar /maven/docker-demo.jar</cmd>
</run>
```

5.2.2. Environment and Labels

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

Example

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

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

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

specified in this file override any environment variables specified in the configuration.

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

Example

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

5.2.3. Port Mapping

The `<ports>` configuration contains a list of port mappings. Each mapping has multiple parts, each separate by a colon. This is equivalent to the port mapping when using the Docker CLI with option `-p`.

A `port` stanza may take one of the following forms:

Table 12. Port mapping format

Format	Description
18080:8080	Tuple consisting of two numeric values separated by a `:`. This form will result in an explicit mapping between the docker host and the corresponding port inside the container. In the above example, port 18080 would be exposed on the docker host and mapped to port 8080 in the running container.
***host.port*:80**	Tuple consisting of a string and a numeric value separated by a `:`. In this form, the string portion of the tuple will correspond to a Maven property. If the property is undefined when the `start` task executes, a port will be dynamically selected by Docker in the ephemeral port range and assigned to the property which may then be used later in the same POM file. The ephemeral port range is configured by the `/proc/sys/net/ipv4/ip_local_port_range` kernel parameter, which typically ranges from 32678 to 61000. If the property exists and has numeric value, that value will be used as the exposed port on the docker host as in the previous form. In the above example, the docker service will elect a new port and assign the value to the property `host.port` which may then later be used in a property expression similar to `<value>${host.port}</value>`. This can be used to pin a port from the outside when doing some initial testing similar to `mvn -Dhost.port=10080 docker:start`
bindTo:_host.port_:80	Tuple consisting of two strings and a numeric value separated by a `:`. In this form, `bindTo` is an ip address on the host the container should bind to. As a convenience, a hostname pointing to the docker host may also be specified. The container will fail to start if the hostname can not be resolved.

Format	Description
+host.ip:_host.port_:80	Tuple consisting of two strings and a numeric value separated by a `:`. In this form, the host ip of the container will be placed into a Maven property name `host.ip`. If docker reports that value to be `0.0.0.0`, the value of `docker.host.address` will be substituted instead. In the event you want to use this form and have the container bind to a specific hostname/ip address, you can declare a Maven property of the same name (`host.ip` in this example) containing the value to use. `host:port` works in the same way as described above.

The following are examples of valid configuration entries:

Example

```
<properties>
  <bind.host.ip>1.2.3.4</bind.host.ip>
  <bind.host.name>some.host.pvt</bind.host.name>
<properties>

...

<ports>
  <port>18080:8080</port>
  <port>host.port:80</port>
  <port>127.0.0.1:80:80</port>
  <port>localhost:host.port:80</port>
  <port>+container.ip.property:host.port:5678</port>
  <port>+bind.host.ip:host.port:5678</port>
  <port>+bind.host.name:5678:5678</port>
<ports>
```

Another useful configuration option is `portPropertyFile` which can be used to to write out the container's host ip and any dynamic ports that have been resolved. The keys of this property file are the property names defined in the port mapping configuration and their values those of the corresponding docker attributes.

This property file might be useful with tests or with other maven plugins that will be unable to use the resolved properties because they can only be updated after the container has started and plugins resolve their properties in an earlier lifecycle phase.

If you don't need to write out such a property file and thus don't need to preserve the property names, you can use normal maven properties as well. E.g. `${host.var}:${port.var}:8080` instead of `+host.var:port.var:8080`.

5.2.4. Network Links

The `<links>` configuration contains a list of containers that should be linked to this container according to Docker Links. Each link can have two parts where the optional right side is separated by a `:` and will be used as the name in the environment variables and the left side refers to the name of the container linking to. This is equivalent to the linking when using the Docker CLI `--link`

option.

Example for linking to a container with name or alias *postgres* :

Example

```
<links>
  <link>postgres:db</link>
</links>
```

This will create the following environment variables, given that the postgres image exposes TCP port 5432:

Example

```
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5432_TCP=tcp://172.17.0.5:5432
DB_PORT_5432_TCP_PROTO=tcp
DB_PORT_5432_TCP_PORT=5432
DB_PORT_5432_TCP_ADDR=172.17.0.5
```

If you wish to link to existing containers not managed by the plugin, you may do so by specifying the container name obtained via `docker ps` in the configuration.

Please note that the link behaviour also depends on the network mode selected. Links as described are referred to by Docker as *legacy links* and might vanish in the future. For custom networks no environments variables are set and links create merely network aliases for the linked container.

For a more detailed documentation for the new link handling please refer to the Docker network documentation

5.2.5. Restart Policy

Specify the behavior to apply when the container exits. These values can be specified withing a `<restartPolicy>` section with the following sub-elements:

Table 13. Restart Policy configuration

Element	Description
name	Restart policy name, choose from: * `always` (*v1.15*) always restart * `on-failure` (*v1.15*) restart on container non-exit code of zero
retry	If `on-failure` is used, controls max number of attempts to restart before giving up.

The behavior to apply when the container exits. The value is an object with a name property of either "always" to always restart or "on-failure" to restart only when the container exit code is non-zero. If on-failure is used, MaximumRetryCount controls the number of times to retry before giving up. The default is not to restart. (optional)

5.2.6. Volumes

A container can bind (or "mount") volumes from various source when starting up: Either from a directory of the host system or from another container which exports one or more directories. The mount configuration is specified within a `<volumes>` section of the run configuration. It can contain the following sub elements:

Table 14. Volume configuration

Element	Description
from	List of `<image>` elements which specify image names or aliases of containers whose volumes should be imported.
bind	List of `<volume>` specifications (or *host mounts*). Use `/path` to create and expose a new volume in the container, `/host_path:/container_path` to mount a host path into the container and `/host_path:/container_path:ro` to bind it read-only.

Volumes example

```
<volumes>
  <bind>
    <volume>/logs</volume>
    <volume>/opt/host_export:/opt/container_import</volume>
  </bind>
  <from>
    <image>jolokia/docker-demo</image>
  </from>
</volumes>
```

In this example the container creates a new volume named `/logs` on the container and mounts `/opt/host_export` from the host as `/opt/container_import` on the container. In addition all exported volumes from the container which has been created from the image `jolokia/docker-demo` are mounted directly into the container (with the same directory names under which the exporting container exposes these directories). This image must be also configured for this plugin. Instead of the full image name, an alias name can be used, too.

Please note, that no relative paths are allowed. However, you can use Maven variables in the path specifications. This should even work for boot2docker and docker-machine:

Example with absolute paths

```
<volumes>
  <bind>
    <volume>${project.build.directory}/${project.artifactId}-
${project.version}:/usr/local/tomcat/webapps/${project.name}</volume>
    <volume>${project.basedir}/data:/data</volume>
  </bind>
</volumes>
```

If you wish to mount volumes from an existing container not managed by the plugin, you may do

by specifying the container name obtained via `docker ps` in the configuration.

5.2.7. Wait

While starting a container is it possible to block the execution until some condition is met. These conditions can be specified within a `<wait>` section which the following sub-elements:

Table 15. Wait configuration

Element	Description
http	HTTP ping check which periodically polls an URL. It knows the following sub-elements: * **url** holds an URL and is mandatory * **method** Optional HTTP method to use. * **status** Status code which if returned is considered to be a successful ping. This code can be given either as a single number (200) or as a range (200..399). The default is `200..399`
log	Regular expression which is applied against the log output of an container and blocks until the pattern is matched.
time	Time in milliseconds to block.
kill	Time in milliseconds between sending `SIGTERM` and `SIGKILL` when stopping a container. Since docker itself uses second granularity, you should use at least 1000 milliseconds.
shutdown	Time to wait in milliseconds between stopping a container and removing it. This might be helpful in situation where a Docker croaks with an error when trying to remove a container to fast after it has been stopped.
exec	Commands to execute during specified lifecycle of the container. It knows the following sub-elements: * **postStart** Command to run after the above wait criteria has been met * **preStop** Command to run before the container is stopped.
tcp	TCP port check which periodically polls given tcp ports. It knows the following sub-elements: * **mode** can be either `mapped` which uses the mapped ports or `direct` in which case the container ports are addressed directly. In the later case the host field should be left empty in order to select the container ip (which must be routed which is only the case when running on the Docker daemon's host directly). Default is `direct` when host is *localhost*, `mapped` otherwise. The direct mode might help when a so called *user-proxy* is enabled on the Docker daemon which makes the mapped ports directly available even when the container is not ready yet. * **host** is the hostname or the IP address. It defaults to `${docker.host.address}` for a mapped mode and the container ip address for the direct mode. * **ports** is a list of TCP ports to check. These are supposed to be the container internal ports.

As soon as one condition is met the build continues. If you add a `<time>` constraint this works more or less as a timeout for other conditions. The build will abort if you wait on an url or log output and reach the timeout. If only a `<time>` is specified, the build will wait that amount of milliseconds and then continues.

Example

```
<wait>
  <http>
    <url>http://localhost:${host.port}</url>
    <method>GET</method>
    <status>200..399</status>
  </http>
  <time>10000</time>
  <kill>1000</kill>
  <shutdown>500</shutdown>
  <exec>
     <postStart>/opt/init_db.sh</postStart>
     <preStop>/opt/notify_end.sh</preStop>
  </exec>
  <tcp>
     <host>192.168.99.100</host>
     <ports>
        <port>3306</port>
        <port>9999</port>
     </ports>
  </tcp>
</wait>
```

This setup will wait for the given URL to be reachable but ten seconds at most. Additionally, it will be waited for the TCP ports 3306 and 9999. Also, when stopping the container after an integration tests, the build wait for 500 ms before it tries to remove the container (if not `keepContainer` or `keepRunning` is used). You can use maven properties in each condition, too. In the example, the `${host.port}` property is probably set before within a port mapping section.

The property `${docker.host.address}` is set implicitly to the address of the Docker host. This host will be taken from the `docker.host` configuration if HTTP or HTTPS is used. If a Unix socket is used for communication with the docker daemon, then `localhost` is assumed. You can override this property always by setting this Maven property explicitly.

5.2.8. Logging

When running containers the standard output and standard error of the container can be printed out. Several options are available for configuring the log output:

Table 16. Logging configuration

Element	Description
enabled	If set to `false` log output is disabled. This is useful if you want to disable log output by default but want to use the other configuration options when log output is switched on on the command line with `-Ddocker.showLogs`. Logging is enabled by default if a `<log>` section is given.
prefix	Prefix to use for the log output in order to identify the container. By default the image `alias` is used or alternatively the container `id`.

Element	Description
date	Dateformat to use for log timestamps. If `<date>` is not given no timestamp will be shown. The date specification can be either a constant or a date format. The recognized constants are: * `NONE` Switch off timestamp output. Useful on the command line (`-Ddocker.logDate=NONE`) for switching off otherwise enabled logging. * `DEFAULT` A default format in the form `HH:mm:ss.SSS` * `MEDIUM` Joda medium date time format * `SHORT` Joda short date time format * `LONG` Joda long date time format * `ISO8601` Full ISO-8601 formatted date time with milli seconds As an alternative a date-time format string as recognized by JodaTime is possible. In order to set a consistent date format the global configuration parameter `logDate` can be used.
color	Color used for coloring the prefix when coloring is enabeld (i.e. if running in a console and `useColor` is set). The available colors are `YELLOW`, `CYAN`, `MAGENTA`, `GREEN`, `RED`, `BLUE`. If coloring is enabled and now color is provided a color is picked for you.
file	Path to a file to which the log output is written. This file is overwritten for every run and colors are switched off.
driver	Section which can specify a dedicated log driver to use. A `<name>` tag within this section depicts the logging driver with the options specified in `<opts>`. See the example below for how to use this.

Example

```
<log>
  <prefix>TC</prefix>
  <date>default</date>
  <color>cyan</color>
</log>
```

The following example switches on the `gelf` logging driver . This is equivalent to the options `--log-driver=gelf --log-opt gelf-address=udp://localhost:12201` when using `docker run`.

```
<log>
  ...
  <driver>
    <name>gelf</name>
    <opts>
      <gelf-address>udp://localhost:12201</gelf-address>
    </opts>
  </driver>
</log>
```

5.3. docker:stop

Stops and removes a docker container. This goals stops every container started with `<docker:start>` either during the same build (e.g. when bound to lifecycle phases when doing integration tests) or

for containers created by a previous call to `<docker:start>`

If called within the same build run, only the containers that were explicitly started during the run will be stopped. Existing containers started using `docker:start` for the project will not be affected.

If called as a separate invocation, the plugin will stop and remove any container it finds whose image is defined in the project's configuration. Any existing containers found running whose image name matches but was not started by the plugin will not be affected.

In case the naming strategy for an image is `alias` (i.e. the container name is set to the given alias), then only the container with this alias is stopped. Other containers originating from the same image are not touched.

It should be noted that any containers created prior to version `0.13.7` of the plugin may not be stopped correctly by the plugin because the label needed to tie the container to the project may not exist. Should this happen, you will need to use the Docker CLI to clean up the containers and/or use the `docker.sledgehammer` option listed below.

For tuning what should happen when stopping there are four global parameters which are typically used as system properties:

Table 17. Stop configuration

Element	Description	Parameter
keepContainer	If set to `true` not destroy container after they have been stopped. Default is false.	`docker.keepContainer`
keepRunning	If set to `true` actually don't stop the container. This apparently makes only sense when used on the command line when doing integration testing (i.e. calling `docker:stop` during a lifecycle binding) so that the container are still running after an integration test. This is useful for analysis of the containers (e.g. by entering it with `docker exec`).	`docker.keepRunning`
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`
allContainers	Stops and removes any container that matches an image defined in the current project's configuration. This was the default behavior of the plugin prior up to version 0.13.6	`docker.allContainers`

Example

```
$ mvn -Ddocker.keepRunning clean install
```

5.4. docker: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 18. 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`

5.5. docker:watch

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

`docker: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 docker:build docker: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 `docker: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 `docker:start`.

Example

```
$ mvn docker:start docker: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`.

`docker: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 19. 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`

Element	Description	Property
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 `docker:watch` has been stopped. By default this is set to `false`.	`docker.keepRunning`
keepContainer	As for `docker: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 20. 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.
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 docker:build docker:start docker: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.

5.6. docker:remove

This goal can be used to clean up images and containers. By default all so called *data images* are removed with its containers. A data image is an image without a run configuration. This can be tuned by providing the properties `removeAll` which indicates to remove all images managed by this build. As with the other goals, the configuration `image` can be used to tune the images to remove. All containers belonging to the images are removed as well.

Considering three images 'db','tomcat' and 'data' where 'data' is the only data images this example demonstrates the effect of this goal:

- `mvn docker:remove` will remove 'data'
- `mvn -Ddocker.removeAll docker:remove` will remove all three images
- `mvn -Ddocker.image=data,tomcat docker:remove` will remove 'data'
- `mvn -Ddocker.image=data,tomcat -Ddocker.removeAll docker:remove` will remove 'data' and 'tomcat'

5.7. docker:logs

With this goal it is possible to print out the logs of containers started from images configured in this plugin. By default only the latest container started is printed, but this can be changed with a property. The format of the log output is influenced by run configuration of the configured images. The following system properties can the behaviour of this goal:

Table 21. Logging options

Property	Description
docker.logAll	If set to `true` the logs of all containers created from images configured for this plugin are printed. The container id is then prefixed before every log line. These images can contain many containers which are already stopped. It is probably a better idea to use `docker logs` diretly from the command line.
docker.follow	If given will wait for subsequent log output until CRTL-C is pressed. This is similar to the behaviour of `docker logs -f` (or `tail -f`).
docker.image	Filte to restrict the set of images for which log should be fetched. This can be a comma separated list of image or alias names.
docker.logDate	Date format to use. See "Logging" for available formats.

Example

```
$ mvn docker:logs -Ddocker.follow -Ddocker.logDate=DEFAULT
```

5.8. docker:source

The `docker:source` target can be used to attach a docker build archive containing the Dockerfile and all added files to the Maven project with a certain classifier. It reuses the configuration from docker:build.

`docker:source` uses the image's alias as part of the classifier, so it is mandatory that the alias is set for this goal to work. The classifier is calculated as `docker-<alias>` so when the alias is set to `service`, then the classifier is `docker-service`.

`docker:source` can be attached to a Maven execution phase, which is `generate-sources` by default.

For example, this configuration will attach the docker build archive to the artifacts to store in the repository:

Example

```
<plugin>
  <artifactId>docker-maven-plugin</artifactId>
  <!-- ..... -->
  <executions>
     <execution>
       <id>sources</id>
       <goals>
         <goal>source</goal>
       </goals>
     </execution>
  </executions>
</plugin>
```

If the plugin is not bound to an execute phase but called directly, it must be ensured that the package phase is called, too. That is required to find the artifacts created with this build when referenced from an assembly.

Chapter 6. External Configuration

For special configuration needs there is the possibility to get the runtime and build configuration from places outside the plugin's configuration. This is done with the help of `<external>` configuration sections which at least has a `<type>` subelement. This `<type>` element selects a specific so called "handler" which is responsible for creating the full image configuration. A handler can decided to use the `<run>` and `<build>` configuration which could be provided in addition to this `<external>` section or it can decide to completely ignore any extra configuration option.

A handler can also decide to expand this single image configuration to a list of image configurations. The image configurations resulting from such a external configuration are added to the *regular* `<image>` configurations without an `<external>` section.

The available handlers are described in the following.

6.1. Property based configuration

For simple needs the image configuration can be completely defined via Maven properties which are defined outside of this plugin's configuration. Such a property based configuration can be selected with an `<type>` of `props`. As extra configuration a prefix for the properties can be defined which by default is `docker`.

Example

```
<image>
  <external>
     <type>props</type>
     <prefix>docker</prefix> <!-- this is the default -->
  </external>
</image>
```

Given this example configuration a single image configuration is built up from the following properties, which correspond to the corresponding values in the `<build>` and `<run>` sections.

Table 22. External properties

docker.alias	Alias name
docker.args.BUILDVAR	Set the value of a build variable. The syntax is the same as for specifying environment variables (see below).
docker.assembly.baseDir	Directory name for the exported artifacts as described in an assembly (which is `/maven` by default).
docker.assembly.descriptor	Path to the assembly descriptor when building an image
docker.assembly.descriptorRef	Name of a predefined assembly to use.

docker.assembly.exportBaseDir	If `true` export base directory
docker.assembly.ignorePermissions	If set to `true` existing file permissions are ignored when creating the assembly archive. Deprecated, use a permission mode of `ignore` instead.
docker.assembly.permissions	can be `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`) or `auto` to let the plugin select `exec` on Windows and `keep` on others. `keep` is the default value.
docker.assembly.dockerFileDir	specifies a directory containing an external Dockerfile that will be used to create the image. This is deprecated p ease use `docker.dockerFileDir` or `docker.dockerFile` instead.
docker.nocache	Don't use Docker's build cache.This can be overwritten by setting a system property `docker.nocache` when running Maven.
docker.bind.idx	Sets a list of paths to bind/expose in the container
docker.capAdd.idx	List of kernel capabilities to add to the container
docker.capDrop.idx	List of kernel capabilities to remove from the container
docker.cleanup	Cleanup dangling (untagged) images after each build (including any containers created from them). Default is `try` (which wont fail the build if removing fails), other possible values are `none` (no cleanup) or `remove` (remove but fail if unsuccessful)
docker.cmd	Command to execute. This is used both when running a container and as default command when creating an image.
docker.domainname	Container domain name
docker.dns.idx	List of dns servers to use
docker.dnsSearch.idx	List of dns search domains
docker.dockerFile	specifies a Dockerfile to use. This property must point to the Dockerfile itself.
docker.dockerFileDir	specifies a directory containing an external dockerfile that will be used to create the image. The dockerfile must be name `Dockerfile`
docker.entrypoint	Container entry point
docker.workdir	Container working directory

docker.env.VARIABLE	Sets an environment variable. E.g. `<docker.env.JAVA_OPTS>-Xmx512m</docker.env.JAVA_OPTS>` sets the environment variable `JAVA_OPTS`. Multiple such entries can be provided. This environment is used both for building images and running containers. The value cannot be empty but can contain Maven property names which are resolved before the Dockerfile is created
docker.envPropertyFile	specifies the path to a property file whose properties are used as environment variables. The environment variables takes precedence over any other environment variables specified.
docker.extraHosts.idx	List of `host:ip` to add to `/etc/hosts`
docker.from	Base image for building an image
docker.hostname	Container hostname
docker.labels.LABEL	Sets a label which works similarly like setting environment variables.
docker.log.enabled	Use logging (default: `true`)
docker.log.prefix	Output prefix
docker.log.color	ANSI color to use for the prefix
docker.log.date	Date format for printing the timestamp
docker.log.driver.name	Name of an alternative log driver
docker.log.driver.opts.VARIABLE	Logging driver options (specified similar as in `docker.env.VARIABLE`)
docker.links.idx	defines a list of links to other containers when starting a container. *idx* can be any suffix which is not use except when *idx* is numeric it specifies the order within the list (i.e. the list contains first a entries with numeric indexes sorted and the all non-numeric indexes in arbitrary order). For example `<docker.links.1>db</docker.links.1>` specifies a link to the image with alias 'db'.
docker.maintainer	defines the maintainer's email as used when building an image
docker.memory	Container memory (in bytes)
docker.memorySwap	Total memory (swap + memory) `-1` to disable swap
docker.name	Image name

docker.namingStrategy	Container naming (either `none` or `alias`)
docker.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.
docker.portPropertyFile	specifies a path to a port mapping used when starting a container.
docker.ports.idx	Sets a port mapping. For example `<docker.ports.1>jolokia.ports:8080<docker.ports.1>` maps the container port 8080 dynamically to a host port and assigns this host port to the Maven property `${jolokia.port}`. See Port mapping for possible mapping options. When creating images images only the right most port is used for exposing the port. For providing multiple port mappings, the index should be count up.
docker.registry	Registry to use for pushing images.
docker.restartPolicy.name	Container restart policy
docker.restartPolicy.retry	Max restrart retries if `on-failure` used
docker.tags.idx	defines a list of tags to apply to a built image
docker.ulimits.idx	Ulimits for the container. Ulimit is specified with a soft and hard limit `<type>=<soft limit>[:<hard limit>]`. For example `docker.ulimits.1=memlock=-1:-1`
docker.user	User to switch to at the end of a Dockerfile. Not to confuse with `docker.username` which is used for authentication when interacting with a Docker registry.
docker.volumes.idx	defines a list of volumes to expose when building an image
docker.volumesFrom.idx	defines a list of image aliases from which the volumes should be mounted of the container. The list semantics is the same as for links (see above). For examples `<docker.volumesFrom.1>data</docker.volumesFrom.1>` will mount all volumes exported by the `data` image.
docker.wait.http.url	URL to wait for during startup of a container
docker.wait.http.method	HTTP method to use for ping check
docker.wait.http.status	Status code to wait for when doing HTTP ping check
docker.wait.time	Amount of time to wait during startup of a container (in ms)
docker.wait.log	Wait for a log output to appear.

docker.wait.exec.postStart	Command to execute after the container has start up.
docker.wait.exec.preStop	Command to execute before command stops.
docker.wait.shutdown	Time in milliseconds to wait between stopping a container and removing it.
docker.wait.tcp.mode	Either `mapped` or `direct` when waiting on TCP connections
docker.wait.tcp.host	Hostname to use for a TCP wait checks
docker.wait.tcp.port.idx	List of ports to use for a TCP check.
docker.wait.kill	Time in milliseconds to wait between sending SIGTERM and SIGKILL to a container when stopping it.
docker.workingDir	Working dir for commands to run in

Any other `<run>` or `<build>` sections are ignored when this handler is used. Multiple property configuration handlers can be used if they use different prefixes. As stated above the environment and ports configuration are both used for running container and building images. If you need a separate configuration you should use explicit run and build configuration sections.

Example

```
<properties>
  <docker.name>jolokia/demo</docker.name>
  <docker.alias>service</docker.alias>
  <docker.from>consol/tomcat:7.0</docker.from>
  <docker.assembly.descriptor>src/main/docker-
assembly.xml</docker.assembly.descriptor>
  <docker.env.CATALINA_OPTS>-Xmx32m</docker.env.CATALINA_OPTS>
  <docker.label.version>${project.version}</docker.label.version>
  <docker.ports.jolokia.port>8080</docker.ports.jolokia.port>
  <docker.wait.url>http://localhost:${jolokia.port}/jolokia</docker.wait.url>
</properties>

<build>
  <plugins>
    <plugin>
      <groupId>io.fabric8</groupId>
      <artifactId>docker-maven-plugin</artifactId>
      <configuration>
        <images>
          <image>
            <external>
              <type>props</type>
              <prefix>docker</prefix>
            </external>
          </image>
        </images>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Chapter 7. 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 `docker: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 docker:build docker: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 8. Authentication

When pulling (via the `autoPull` mode of `docker: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 docker: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.

8.1. 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.

8.2. 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.

8.3. 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 9. Further reading

- Examples:
  - Examples are below `samples/` and contain example setups which you can use as blueprints for your own projects.
  - A Shootout for comparing docker maven plugins
  - Another sample project with a Microservice and a Database.
- ChangeLog has the release history of this plugin.
- Contributing explains how you can contribute to this project. Pull requests are highly appreciated!